Construction de systèmes repartis securises à base de composants

Lilia Sfaxi

UNIVERSITE DE TUNIS EL MANAR
FACULTE DES SCIENCES DE TUNIS

UNIVERSITÉ DE GRENOBLE

FORMATION DOCTORALE EN INFORMATIQUE

THESE

présentée en vue de l'obtention du
Doctorat en Informatique

par

Lilia SFAXI-YOUSSEF
(Mastère en Micro et Nano-Electronique, INP Grenoble)

Construction de systèmes répartis sécurisés à base de composants

soutenue le 05 mai 2012, devant le jury d'examen

MM.	**Khaled BSAÏES**	**Professeur (Univ. de Tunis El Manar, Tunisie), Président**
	Belhassen ZOUARI	**Professeur (Univ. de Carthage), Rapporteur**
	Mohamed MOSBAH	**Professeur (Univ. de Bordeaux I), Rapporteur**
	Didier DONSEZ	**Professeur (Univ. de Grenoble), Membre**
	Yassine LAKHNECH	**Professeur (Univ. de Grenoble), Directeur de thèse**
	Riadh ROBBANA	**Professeur (INSAT-Tunis, Tunisie), Directeur de thèse**

Thèse préparée sous convention de cotutelle
FST-Tunis, Tunisie (Laboratoire LIP2) et Université de Grenoble, France (Laboratoire VERIMAG)

Dédicace

Finir une thèse, c'est un peu comme marier son enfant : il a beau être sorti de la maison, on continue à y penser et à penser à son bien-être.

Alors merci à tous ceux qui m'ont aidé à l'élever et à le nourrir, qui ont été là pendant mes périodes de fatigue, de déprime ou de bonheur. Merci à tous ceux que je porte secrètement ou ouvertement dans mon cœur et qui me le retournent sans aucune restriction.

A mes parents, qui ont continué à penser à mon bien-être. Que Dieu vous garde le plus longtemps possible, et vous préserve de toutes les difficultés de la vie. Dieu seul sait comme je vous aime, même si je ne le montre pas assez souvent.

A mon mari, sans qui je ne serai pas où j'en suis. Tu es ce qui m'est arrivé de meilleur, et très sincèrement, le choix le plus intelligent que j'ai fait de ma vie !

A ma petite sœur, que j'aime plus que tout. Je te souhaite tout le bonheur du monde, car tu le mérites.

A mon petit frère, mon soleil. Je suis fière de toi, de ce que tu es et de ce que deviens.

A mes tantes : Aziza, Boutheina, Gihene, Malek et Monia. Je dis toujours que j'ai le bonheur d'avoir non pas une mère, mais SIX, avec tout ce que ça apporte de joie, mais aussi de pression ! Merci d'avoir été là pour moi, à tous moments, et j'espère vous rendre aussi fières de moi que je suis fière de vous.

A ma nouvelle famille : Habib, Souad, Imen, Wajih, Ines et Amenallah. Merci de m'avoir adopté, de m'avoir aimé et de m'avoir soutenue.

A tous les membres des familles SFAXI, SOHLOBJI et YOUSSEF.

A mes amis d'enfance, d'adolescence, de jeunesse, d'hier et d'aujourd'hui.

A ma Tunisie.

Remerciements

Je tiens à exprimer mes remerciements à toutes les personnes qui ont plus ou moins directement contribué à l'accomplissement de cette thèse.

J'aimerai d'abord remercier mes directeurs de thèse : le Pr. Riadh ROBBANA de l'université de Carthage pour ses conseils, sa présence continuelle et son aide précieuse, autant du côté scientifique que du côté humain ; et le Pr. Yassine LAKHNECH de l'université de Grenoble de m'avoir fait confiance et accueilli au sein de son équipe. Je le remercie pour ses conseils et son aide.

J'aimerai remercier ma co-encadrante, Dr Takoua ABDELLATIF de m'avoir guidé, conseillé et suivi. Son grand sérieux, sa compétence et sa rigueur m'ont beaucoup aidé, tout au long de cette thèse.

Je remercie également Pr. Mohamed MOALLA, directeur du laboratoire LIP2, de son engagement, son aide et ses conseils.

Merci à Pr. Khaled BSAIES, professeur à la Faculté des Sciences de Tunis, de m'avoir fait l'honneur de présider mon jury de thèse.

Je remercie Pr. Belhassen ZOUARI, professeur à la SupCOM et Pr. Mohamed MOSBAH, professeur à l'Institut Polytechnique de Bordeaux d'avoir accepté d'être les rapporteurs de mes travaux de thèse. Je les remercie aussi pour leurs conseils en vue de l'amélioration de ce manuscrit.

Merci à Pr. Didier DONSEZ, professeur à l'Université de Grenoble d'avoir accepté de faire partie de mon jury de thèse.

Je profite aussi de cet espace pour exprimer mes remerciements à toutes les personnes que j'ai côtoyées pendant ces années de thèse, notamment les membres respectifs du laboratoire Verimag et du laboratoire LIP2. Je remercie particulièrement Mme Sandrine MAGNIN du laboratoire Verimag et Mme Alice Corrazzini de l'école doctorale MSTII d'avoir montré beaucoup de patience avec moi, et d'avoir facilité grandement mes communications à distance avec les deux établissements.

Un grand merci à toute ma famille, à mon mari et à mes amis de simplement avoir été là pour moi.

Table des matières

Liste des Figures

Liste des Tableaux

Introduction Générale

Tout l'intérêt de l'art se trouve dans le commencement.
Après le commencement, c'est déjà la fin.
[Pablo Picasso]

Introduction Générale

Dans cette ère de calcul massif, de services informatiques sur internet et de l'informatique en nuage (ou *cloud computing*), les systèmes distribués sont de plus en plus populaires. Ces systèmes sont composés d'un ensemble de machines indépendantes qui agissent −de manière transparente à l'utilisateur− comme un système unique. Ils offrent beaucoup d'avantages : un gain non négligeable en puissance de calcul, en capacité de stockage et en performance, grâce à un traitement parallèle, et une grande extensibilité, car ils peuvent croître progressivement selon le besoin. Cependant, leur flexibilité et évolutivité soulèvent des problèmes supplémentaires, dont notamment la sécurité. En effet, Lamport définit ces systèmes comme étant des systèmes "*qui vous empêchent de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne.*" Étant donné qu'une information dans un système distribué peut être manipulée, échangée, dupliquée et modifiée par des entités qui évoluent dans des environnements hétérogènes et parfois peu fiables, il est impératif de trouver un moyen de contrôler leur utilisation de manière transparente et surtout peu importune pour l'utilisateur.

Cependant, à l'exception des procédure de contrôle d'accès souvent intuitives, il est rare de trouver une stratégie de sécurité bien établie et claire dans le développement des logiciels distribués. Pourtant, ces logiciels manipulent des informations ou assurent des services souvent sensibles qui requièrent une attention particulière. Il est donc déconseillé aux concepteurs de se contenter de mécanismes de sécurité peu rigoureux.

Les mécanismes de sécurité classiques sont la plupart du temps axés sur les politiques de contrôle d'accès et l'utilisation de primitives cryptographiques. Cependant, ces mécanismes ne permettent pas de mettre en œuvre certaines propriétés de sécurité plus avancées, notamment la non-interférence. La non-interférence est une propriété définie initialement par Goguen et Meseguer[Goguen82], qui assure que les données sensibles n'affectent pas le comportement publiquement visible du système. Cette propriété permet le suivi de l'information dans le système,

et l'application de la confidentialité et de l'intégrité de bout en bout. Par exemple, une politique de contrôle d'accès peut requérir que seuls les utilisateurs ayant les droits en lecture appropriés peuvent lire un fichier *f* ; alors que la non-interférence exigera qu'aucun des utilisateurs n'ayant pas le niveau de sécurité approprié ne peut avoir une quelconque information sur le contenu du fichier *f*, même indirectement. Une définition formelle de la non-interférence sera présentée dans la section III.1.1, partie I.

Les mécanismes de contrôle de flux d'information (CFI)[Denning77, Goguen82, Bell75, Biba77] sont devenus classiques pour résoudre le problème de non-interférence. Le contrôle de flux d'information se base sur le suivi de la propagation de l'information dans le système, en attribuant des étiquettes de sécurité aux différents canaux de communication du système. Une étiquette représente un niveau de sécurité déterminé et permet de classer les différents canaux de communication selon leur niveau de restriction. Considérons un système qui traite des données provenant des canaux d'entrée et qui les diffuse en utilisant des canaux de sortie. Nous attribuons des étiquettes à ces canaux, par exemple l'étiquette *High* pour les canaux manipulant des informations sensibles ou secrètes, et *Low* pour les canaux manipulant des informations publiques. La propriété de non-interférence implique que, pendant l'exécution, les données provenant de canaux d'entrée de niveau *High* ne doivent pas circuler vers des canaux de sortie de niveau *Low*, car en observant le comportement des sorties publiques, un attaquant peut déduire la valeur des entrées sensibles.

La recherche sur les politiques de contrôle de flux d'information en général a débuté depuis les années 70, et celle sur la non-interférence au début des années 80. Néanmoins, les solutions proposées ne sont pas encore exploitées dans des systèmes réels et n'ont pas vraiment abouti à des produits dans l'industrie, cela étant dû au fait que l'implémentation de la non-interférence reste difficile à mettre en œuvre pour des systèmes réels[Zdancewic04]. Les systèmes récemment construits, basés sur la modification des systèmes d'exploitation (comme Flume [Krohn07], HiStar [Zeldovich06] et Asbestos[Efstathopoulos05]) ou l'utilisation d'annotations dans les programmes comme le langage JIF[Myers00] ne sont pas très efficaces pour la construction de systèmes sécurisés. En effet, les approches dynamiques au niveau du système d'exploitation ne permettent pas de suivre les flux internes des applications, puisqu'elles n'observent que les entrées et sorties des processus. Étant donné qu'elles considèrent les flux d'information à un niveau de granularité assez grossier, elles peuvent provoquer des sur-approximations à l'origine de faux-positifs : le système peut détecter des interférences alors qu'elles n'existent pas.

D'un autre côté, l'approche statique du contrôle de flux, basée sur les langages typés comme JIF, permet un contrôle de flux d'information à grain plus fin. Toutefois, ces langages sont compliqués à utiliser : les développeurs sont obligés de propager les étiquettes manuellement dans tout le code. Ils doivent surtout être à même de savoir attribuer les étiquettes adéquates

à toutes les variables intermédiaires, méthodes, exceptions, etc. En pratique, nous avons besoin d'attribuer des niveaux de sécurité à quelques données et non à la totalité du code. De plus, la configuration des politiques de sécurité est en général effectuée par les administrateurs des systèmes et non par les développeurs ; ces politiques doivent par conséquent être indépendantes du code.

Il serait donc intéressant de combiner ces deux approches tout en palliant leurs inconvénients : créer un intergiciel (*middleware*) qui puisse vérifier la non-interférence des applications distribuées à la compilation et à l'exécution, en réalisant un contrôle de flux d'information à grain très fin, et ce tout en fournissant un mécanisme qui permette d'appliquer la politique de sécurité aux différents éléments du système à un haut niveau d'abstraction, et indépendamment du code fonctionnel.

Cela peut être grandement facilité par un choix judicieux du modèle de représentation des systèmes distribués qui puisse garantir à la fois la dynamicité, la modularité, la transparence et l'interopérabilité de ces types de systèmes. De plus, une séparation nette entre les nœuds permettra de les configurer séparément. Le modèle le plus approprié pour cela est le modèle à base de composants CBSE (*Component-based Software Engineering*).

Ce modèle décrit les nœuds du système par des composants distincts, ayant chacun une interface pour communiquer avec le monde extérieur, et échangeant des messages via des canaux de communication bien définis. L'utilisation du modèle à base de composants facilite l'intégration de composants hétérogènes dans le même système, et favorise l'utilisation de composants déjà existants ou dont l'implémentation est une boîte noire pour l'utilisateur.

La vérification de la non-interférence étant basée sur le contrôle de flux d'information, son application à des composants permet de distinguer deux types de flux : les flux **internes**, représentés par les circuits d'acheminement des information à l'intérieur d'un composant, et les flux **externes** circulant via les canaux de communication entre les composants. La différence principale entre ces deux flux, c'est qu'un flux externe doit prendre en considération le réseau parfois peu sûr dans lequel il circule, contrairement au flux interne, à l'abri à l'intérieur du composant. Mais les différences s'arrêtent là. Pour ces deux types de flux, la propriété de non-interférence s'applique de la même manière : si nous ne voulons pas que les sorties publiques révèlent des informations sur les entrées secrètes, il faut empêcher les données secrètes de transmettre des informations aux données publiques, et ce *pour chaque flux d'information*, qu'il soit interne ou externe.

L'application de cette contrainte permet de fournir des garanties solides que la confidentialité et l'intégrité des information sont maintenues dans le système de bout en bout. Toutefois, garantir que nous pouvons réaliser un système parfaitement non-interférent est tout simplement impossible, et parfois même indésirable. Cela est dû à plusieurs raisons[Zdancewic03] :

- Parfois, la politique désirée autorise intentionnellement une petite quantité de flux d'information sous réserve de certaines contraintes. Par exemple, Alice peut vouloir divulguer ses données privées à Bob une fois qu'il ait payé pour les avoir, mais pas avant.
- La politique peut exiger que l'information soit gardée secrète pour une période de temps bien déterminée. Par exemple, un service de vente aux enchères en ligne doit publier la valeur de l'offre gagnante une fois l'enchère terminée.
- La cadence avec laquelle une information est intentionnellement révélée peut être considérée trop lente pour être une menace à la sécurité. Par exemple, une fonction de vérification de mot de passe d'un système d'exploitation gère les mots de passe confidentiels, mais même l'interdiction d'accès révèle une petite quantité d'information sur le mot de passe correct. Or cette fuite d'information peut être considérée négligeable, et donc autorisée.
- La non-interférence est la politique désirée, mais l'analyse du programme n'est pas capable de justifier la sécurité de certaines opérations. Par exemple, la fonction de chiffrement d'une librairie de cryptographie prend des données confidentielles et les rend publiques, mais elle garantit la confidentialité de l'information.

Pour permettre ces quelques infractions (légales) à la non-interférence, il faut fournir un mécanisme qui puisse relaxer les contraintes de la politique, souvent jugée trop stricte.

D'autre part, les systèmes distribués utilisent très souvent des composants existants, parfois importés à partir d'autres systèmes sous forme de boîtes noires. Ces composants, qu'on appelle *patrimoniaux*, peuvent constituer de sérieuses failles à la sécurité du système, car rien ne prouve qu'ils respectent la politique de sécurité du système. Il serait donc souhaitable, voire indispensable, de penser à une solution qui fasse en sorte de respecter la confidentialité du code des composants patrimoniaux, tout en contrôlant leur comportement et vérifiant s'il respecte la politique de sécurité désirée.

C'est en prenant en considération ces différents aspects que nous avons réalisé les solutions CIF (*Component Information Flow*) et DCIF (*Distibuted CIF*). CIF offre un ensemble d'outils pour simplifier la configuration et la vérification des propriétés de sécurité pour les systèmes distribués et automatiser leur implantation à la compilation. DCIF est un canevas qui vérifie la non-interférence de ces systèmes à l'exécution. Dans notre approche, nous considérons des systèmes à base de composants répartis sur des machines interconnectées par un réseau qui n'est pas de confiance. L'objectif de CIF et DCIF, c'est qu'un attaquant ne puisse pas inférer ou influencer les valeurs des informations des interfaces de plus haut niveau de sécurité que lui, auxquelles il n'a pas accès directement. La configuration de la politique de sécurité est appliquée au système à un haut niveau d'abstraction et en dehors du code de l'application. Ainsi, non seulement le développeur n'a pas à mélanger l'aspect sécurité avec son code fonctionnel, mais un même code fonctionnel peut être configuré avec des politiques différentes. Les outils vérifient

qu'aucun problème d'interférence n'existe et génèrent les composants cryptographiques qui mettent en œuvre la configuration de la sécurité de haut niveau. Il est également possible de relaxer la propriété de non-interférence de manière quasi-automatique.

Ce manuscrit utilise plusieurs notions que nous devons définir au préalable.

Donnée, information et flux d'information Une **donnée** est un élément brut, qui n'a pas encore d'interprétation ni de contexte. Les données en langage de programmation sont représentées par les variables et les objets.

Une **information** est par définition une donnée interprétée, mise en contexte. Elle crée ainsi une valeur ajoutée pour son récepteur. Par exemple, la variable *mdp* avec sa valeur "azerty" est une donnée. Entrer cette valeur comme étant un mot de passe pour accéder à une application en fait une information dont la connaissance par autrui peut présenter certains dangers pour son propriétaire.

Un **flux d'information** est l'acheminement d'une information d'une donnée à une autre. Ces données peuvent être des variables simples, des attributs ou des ports d'un composant. Le suivi du flux d'information permet de voir quelles sont les différentes entités qui ont eu la possibilité de voir ou de modifier cette information. Cela nous permet de ne pas nous arrêter au niveau de la donnée, mais de suivre l'information en elle même. La concaténation du mot de passe *mdp* avec l'identifiant *login* et sa copie dans une autre variable *loginMdp* représente un flux d'information des données *mdp* et *login* vers *loginMdp*.

Politique, propriété et mécanisme de sécurité D'après l'ITSEC (*Information Technology Security Evaluation Criteria*), une **politique de sécurité** est "l'ensemble des lois, règles et pratiques qui régissent la façon dont l'information sensible et les autres ressources sont gérées, protégées et distribuées à l'intérieur d'un système spécifique". Elle permet de déterminer les objets à sécuriser et identifier les menaces à prendre en compte.

Pour construire une politique de sécurité, il faut définir un ensemble de **propriétés de sécurité** qui doivent être satisfaites par le système. Par exemple, "une information classifiée ne doit pas être transmise à un utilisateur non habilité à la connaître" est une propriété de sécurité.

La politique de sécurité permet de définir l'ensemble des autorités, des ressources et des droits. Affecter des étiquettes de sécurité aux données représente la politique de sécurité désirée pour vérifier la propriété de non-interférence.

Un **mécanisme de sécurité** est le moyen utilisé pour mettre en œuvre la propriété de sécurité. Par exemple, la protection par mot de passe est un mécanisme utilisé pour appliquer la propriété de contrôle d'accès.

Les acteurs du système Dans les systèmes que nous concevons, nous considérons plusieurs types d'acteurs qui interagissent et agissent sur le système. Nous distinguons notamment :

- **Architecte du système** : On l'appelle également **concepteur** du système. Il est chargé de la conception et de la réalisation du plan de construction du système.
- **Développeur** : Cet acteur est responsable de la mise en œuvre du plan conçu par l'architecte en élaborant les algorithmes dans un langage de programmation choisi.
- **Déployeur** : Cet acteur est chargé du déploiement de l'application dans un environnement donné.
- **Administrateur** : C'est le responsable de la configuration et de la mise à jour du système. Il doit avoir une connaissance assez évoluée sur le fonctionnement du système, sur les différents acteurs qui l'utilisent et sur les points faibles qui le caractérisent. C'est l'administrateur qui, entre autres, affecte les contraintes de sécurité au système, applique dessus les mécanismes de sécurité, le teste et intervient si des problèmes ultérieurs interviennent.
- **Attaquant** : Un attaquant peut être tout utilisateur ou application qui utilise l'application protégée et a accès à certaines de ses interfaces en fonction de son niveau de sécurité. Un attaquant peut même être un composant du système qui désire agir sur des informations de niveau de sécurité plus restrictif que le sien. En revanche, il est supposé que l'attaquant ne possède pas la capacité d'inférer de l'information à partir de canaux cachés résultant du temps d'exécution des programmes ou de l'observation de variables internes au programme.

Chacun de ces rôles peut être interprété par une ou plusieurs personnes, et une personne peut assurer plus qu'un rôle. Cependant, pour les systèmes que nous ciblons, la difficulté majeure représente le fait que le développeur de l'application et l'administrateur soient des personnes distinctes. Pour éviter ce problème, nous proposons une plateforme permettant de séparer complètement les aspects de sécurité des aspects fonctionnels, et de permettre l'application des contrôles de sécurité a posteriori par l'administrateur.

Ce manuscrit est composé essentiellement de quatre parties :

- La première partie permet de présenter **le contexte et la problématique** de notre travail, à savoir l'application de la non-interférence statique et dynamique aux systèmes distribués. Nous commençons dans cette partie par présenter les systèmes distribués, les menaces de sécurité auxquelles ils sont sujets, et les mécanismes de sécurité classiques utilisés pour annihiler ces menaces. Nous présentons ensuite le modèle orienté composants que nous allons utiliser pour représenter ce type de systèmes, puis la propriété de non-interférence que nous désirons y appliquer. Nous terminons par citer les difficultés présentes pour appliquer cette propriété sur les systèmes distribués.

- La deuxième partie cite les travaux intéressants et relatifs à notre domaine de recherche dans **l'état de l'art**. Nous avons défini deux grands volets à étudier : les solutions existantes pour résoudre les problèmes de sécurité dans les systèmes distribués, et les solutions d'application de la non-interférence. Nous terminons chaque sous-partie par un comparatif entre les différentes solutions.
- La troisième partie introduit notre **contribution**, notamment les canevas CIF et DCIF. Nous y présentons les architectures de ces solutions, leurs principes et leurs fonctionnements.
- La quatrième partie est une **évaluation** de notre travail. Elle présente en premier lieu un ensemble d'études de cas que nous détaillons pour montrer la faisabilité de notre approche. Elle contient également une évaluation des performances pour le prototype CIF, qui estime le surcout provoqué par l'ajout de l'aspect sécurité au système. Elle se termine enfin par une vérification formelle de notre approche, en appliquant la technique le réécriture pour montrer que notre solution assure bien la non-interférence au niveau du code des composants.

Les travaux de cette thèse ont été publiés dans [Sfaxi10, Sfaxi11, Abdellatif11, Sfaxi11a]. [Sfaxi10] et son extension [Sfaxi11], présentent les outils CIF, appliqués au modèle orienté composants Fractal et au modèle d'étiquettes décentralisé DLM. [Abdellatif11] présente l'architecture des outils CIF, appliqués à la spécification SCA et au modèle à base de jetons, et étayé par une étude de performances. [Sfaxi11a] présente le comportement des outils, et détaille en particulier le comportement de l'algorithme CIFIntra pour des types d'interférences plus variées.

Première partie
Contexte et Problématique : Non-Interférence dans les Systèmes Distribués

Un problème sans solution est un problème mal posé.

[Albert Einstein]

Chapitre I

Sécurité dans les systèmes distribués

Les systèmes distribués ont la particularité de lier plusieurs machines qui cohabitent dans un environnement hétérogène et peu fiable. De plus, leur bon fonctionnement dépend des messages qu'ils se partagent sur le réseau. Il est donc primordial de considérer leur sécurité. Nous présentons dans ce chapitre les menaces de sécurité les plus connues dont souffrent ces systèmes et montrons les différents mécanismes de sécurité existants.

1 Les systèmes distribués

Les systèmes distribués (ou systèmes répartis), en opposition aux systèmes centralisés, sont composés de plusieurs machines indépendantes, ayant des mémoires physiques distinctes, connectées entre elles en réseau et communiquant via ce réseau. Du point de vue de l'utilisateur, aucune différence n'est perceptible entre un système distribué et un système centralisé. Cela dit, ces systèmes permettent de garantir des propriétés qui ne sont pas disponibles dans un système centralisé, comme par exemple la **redondance**, qui permet de pallier les fautes matérielles ou de rendre un même service disponible à plusieurs acteurs sans perte de temps ; la **performance**, garantie par la mise en commun de plusieurs unités de calcul permettant de réaliser des traitements parallélisables en un temps plus court ; et la **protection des données**, qui ne sont pas disponibles partout au même moment, mais dont seules certaines vues sont exportées.

Un système distribué est généralement séparable en plusieurs modules entièrement autonomes, chacun responsable de son propre fonctionnement. Cette autonomie permet d'une part d'utiliser des technologies, plateformes ou langages hétérogènes dans chacun de ces modules, et d'autre part de les exécuter simultanément et garantir ainsi une programmation concurrente.

Cependant, les systèmes distribués sont sujets à plusieurs risques, dus aux points de défaillance qu'ils possèdent en plus des systèmes centralisés, comme par exemple le réseau non sécurisé, les trafics, les nœuds eux-mêmes, etc.

Nous présentons dans les sections suivantes les différentes menaces de sécurité que peuvent rencontrer les systèmes distribués ainsi que les différents mécanismes utilisés pour les éviter.

2 Menaces de sécurité pour les systèmes distribués

Les menaces de sécurité les plus connues pour les systèmes distribués sont causées par des attaques en réseau dont nous citons les plus fréquentes :

DDoS (*Distributed Denial of Service*) Le déni de service distribué est une attaque qui rend le service indisponible pour les utilisateurs en le submergeant de trafic inutile. Cette attaque est provoquée par plusieurs machines à la fois (contrairement au DoS, qui lui est perpétré par un seul attaquant), et est difficile à contrer ou éviter.

MITM (*Man In The Middle*) Cette attaque de l'Homme du Milieu est une forme d'espionnage dans laquelle l'attaquant réalise des connexions indépendantes avec les victimes et relaie les messages entre elles, en leur faisant croire qu'elles sont entrain de discuter entre elles. L'attaquant peut ainsi intercepter tous les messages circulant entre les victimes et en injecter de nouveaux, en se faisant passer respectivement par l'une des victimes auprès de l'autre.

Usurpation d'adresse IP (*IP Spoofing*) C'est une attaque où l'attaquant personnifie une autre machine en envoyant des messages avec son adresse IP.

Reniflement de paquets (*Packet Sniffing*) C'est une attaque où l'attaquant intercepte et enregistre le trafic circulant sur le réseau.

Attaque par rejeu (*Replay Attack*) C'est une attaque de l'Homme du Milieu où l'attaquant répète ou retarde une transmission de données valide. Elle peut être utile pour l'attaquant dans le cas où, par exemple, il désire se faire passer pour un utilisateur en sauvegardant son mot de passe crypté utilisé dans un premier échange comme preuve d'identité, et le renvoyant dans un autre échange.

Ces attaques sont assez fréquentes dans les systèmes d'exploitation, et leur avènement peut s'avérer quelque peu dangereux pour le système, surtout si les données manipulées sont critiques,

comme des données bancaires ou des informations personnelles. Pour les éviter ou les contrer, nous devons veiller à assurer certaines propriétés de sécurité. Nous citons dans cette partie les propriétés les plus importantes.

3 Propriétés de sécurité usuelles

Les propriétés de sécurité les plus connues pour les systèmes distribués sont les suivantes :

Authentification Cette propriété représente la procédure permettant au système distribué de vérifier l'identité d'une entité, que ce soit un utilisateur ou une machine faisant partie ou pas du système, afin d'autoriser son accès à des ressources. Elle permet donc de valider l'authenticité de l'entité en question.

Confidentialité Cette propriété permet de protéger une information dont l'accès est limité aux seules entités admises à la connaître.

Intégrité Cette propriété implique que l'altération de l'information ne peut se faire que dans un cadre volontaire et légitime.

Disponibilité Cette propriété garantit l'aptitude du système à remplir une fonction dans les conditions définies d'horaires, de délai et de performance.

Non répudiation Cette propriété permet d'assurer que l'information ne pourra pas être plausiblement désavouée.

D'autres propriétés plus élaborées sont définies dans la littérature, telle que la propriété de non-interférence [Goguen82] que nous voulons appliquer à nos systèmes. Nous allons l'étudier plus en détails dans le chapitre III, partie I.

4 Mécanismes de sécurité pour les systèmes distribués

Pour pouvoir protéger les systèmes distribués des attaques citées ci-dessus, des mécanismes de sécurité sont définis. Nous les citons dans cette partie.

4.1 Contrôle d'accès

Le contrôle d'accès est un mécanisme de sécurité visant à protéger les ressources physiques en vérifiant si une entité qui demande l'accès à cette ressource a les droits nécessaires pour le faire.

Pour assurer le contrôle d'accès, le système doit fournir à la fois un mécanisme d'**authentification**, qui permet à une entité d'être reconnue par le système (un mot de passe ou une carte, par exemple), et un mécanisme d'**autorisation**, qui permet d'associer à une entité un ensemble de droits sur une ressource.

D'après [Nikander99], le contrôle d'accès inclut le concept de **matrice de contrôle d'accès**, où les colonnes portent les noms des sujets (entités actives), les lignes ceux des objets, et chaque cellule inclut les actions que le sujet est autorisé à réaliser sur l'objet. En pratique, la matrice de contrôle d'accès est un élément abstrait. L'information qui y est incluse est généralement représentée séparément ligne par ligne, sous la forme d'ACLs (*Access Control Lists*), ou colonne par colonne sous la forme de **capacités**(*capabilities*).

4.2 Primitives cryptographiques

Les systèmes distribués ayant cette particularité d'évoluer dans un environnement souvent non fiable, toutes les données échangées entre les nœuds doivent être sécurisées au niveau applicatif. Cette sécurisation pendant le transport implique que (1) les données secrètes ne doivent pas être visibles par un attaquant, (2) l'intégrité des données doit être préservée, dans le sens où les données ne doivent pas être modifiées pendant le transport par des entités tierces et (3) le récepteur de la donnée doit pouvoir vérifier que cette donnée provient bien de l'entité qui prétend l'avoir envoyé. Pour pouvoir garantir ces propriétés, des mécanismes cryptographiques sont généralement utilisés.

Fonction de hachage (*Cryptographic hash*)

Une fonction de hachage est une fonction qui assure l'intégrité d'un message. Elle prend en entrée un message et génère un bloc de bits, de longueur arrivant à plusieurs centaines de bits, qui représente l'**empreinte numérique** du message (*message digest*). Si le message est modifié, même légèrement, par une tierce personne par exemple, un changement significatif est opéré dans l'empreinte (idéalement, 50% de l'empreinte change pour un bit changé dans le message initial).

Plusieurs algorithmes de hachage sont définis. Les deux algorithmes suivants sont les plus utilisés :

- MD5 (*Message Digest 5*) : inventée par Ronald Rivest en 1991, cette fonction est un standard qui produit des empreintes de 128 bits. Le MD5 est une extension plus sécurisée du MD4.
- SHA-1 (*Secure Hash Algorithm-1*) : conçue par la *National Security Agency* (NSA), cette fonction est publiée par le gouvernement des États Unis comme étant un standard fédéral de traitement de l'information. Elle produit un résultat de 160 bits.

Chiffrement

Les empreintes numériques peuvent être utilisées pour assurer l'intégrité d'un message, mais pas la confidentialité. Pour cela, nous avons besoin d'un **chiffrement numérique**. On définit trois types de chiffrement : le chiffrement symétrique, le chiffrement asymétrique et le chiffrement par clef de session.

Chiffrement symétrique Le chiffrement symétrique se fait de la manière suivante : Alice et Bob ont chacun une clef partagée qu'ils sont les seuls à connaître. Ils se mettent d'accord pour utiliser un algorithme cryptographique commun, qu'on appelle *cipher*. Quand Alice veut envoyer un message à Bob, elle chiffre le message originel (le **texte en clair**) pour créer un **cryptogramme**. Elle envoie ensuite le cryptogramme à Bob, qui le reçoit et le déchiffre avec sa clef secrète pour recréer le message en clair originel. Si Chuck est entrain d'épier leur communication, il ne peut voir que le cryptogramme. Ainsi, la confidentialité du message est préservée.

Il est possible de chiffrer bit par bit ou bloc par bloc. Les blocs ont typiquement une taille de 64 bits. Si le message n'est pas un multiple de 64, le dernier bloc (le plus court) doit alors être rempli avec des valeurs aléatoires jusqu'à atteindre les 64 bits (ce concept s'appelle *padding*). Le chiffrement bit par bit est plus utilisé pour les implémentations matérielles.

La force du chiffrement à clef privée est déterminée par l'algorithme de cryptographie et la longueur de la clef.

Il existe plusieurs algorithmes pour le chiffrement à clef privée, dont :

- DES (*Data Encryption Standard*) : inventé par IBM en 1970 et adopté par le gouvernement américain comme standard. C'est un algorithme à blocs de 56 bits.
- **TripleDES** : Utilisé pour gérer les failles des clefs à 56 bits en accroissant la technologie DES en faisant passer le texte en clair à travers l'algorithme DES 3 fois, avec deux clefs différentes, donnant ainsi à la clef une force réelle de 112 bits. Connu également sous le nom de DESede (pour *encrypt, decrypt and encrypt*, les trois phases par lesquelles il passe).

- AES (*Advanced Encryption Standard*) : remplace DES comme standard américain. Il a été inventé par Joan Daemen et Vincent Rijmen et est connu également comme l'algorithme de Rinjdael. C'est un algorithme à blocs de 128 bits avec des clefs de longueur 128, 192 ou 256 bits.
- **Blowfish** : Développé par Bruce Shneider. C'est un algorithme à longueur de clef variable allant de 32 à 448 bits (multiples de 8), et utilisé surtout pour une implémentation sur les logiciels pour les microprocesseurs.
- PBE (*Password Based Encryption*) : algorithme qui utilise le mot de passe comme clef de chiffrement. Il peut être utilisé en combinaison avec une variété d'empreintes numériques et d'algorithmes à clef privée.

Chiffrement asymétrique Le chiffrement symétrique souffre d'un inconvénient majeur : comment partager la clef entre Alice et Bob ? Si Alice la génère, elle doit l'envoyer à Bob ; cependant, c'est une information sensible qui doit être chiffrée. Toutefois, aucune clef n'a été échangée pour réaliser ce chiffrement.

Le chiffrement asymétrique, inventé dans les années 70, a résolu le problème du chiffrement des messages entre deux parties sans accord préalable sur les clefs. Dans ce type de chiffrement, Alice et Bob ont chacun deux paires de clefs différentes : une clef est secrète et ne doit être partagée avec quiconque, et l'autre est publique, et donc visible par tout le monde.

Si Alice veut envoyer un message secret à Bob, elle chiffre le message en utilisant la clef **publique** de Bob et le lui envoie. Bob utilise alors sa clef privée pour déchiffrer le message. Chuck peut voir les deux clefs publiques ainsi que le message chiffré, mais ne peut pas déchiffrer le message parce qu'il n'a pas accès aux clefs secrètes.

Les clefs (publique et secrète) sont générées par paire et sont plus grandes que les clefs de chiffrement privées équivalentes. Il n'est pas possible de déduire une clef à partir de l'autre.

Les deux algorithmes suivants sont utilisés pour le chiffrement à clef publique :

- RSA (*Rivest Shamir Adleman*) : décrit en 1977 par Ronald Rivest, Adi Shamir et Leonard Adleman, et breveté par le MIT (*Massachusets Institute of Technology*) en 1983. C'est l'algorithme de chiffrement à clef publique le plus populaire. Il est très utilisé dans le commerce électronique et l'échange de données sur Internet en général.
- **Diffie-Hellman** : techniquement connu comme un algorithme à *key-agreement* : il ne peut pas être utilisé pour le chiffrement, mais pour permettre aux deux parties de déduire une clef secrète en partageant des informations sur un canal public. Cette clef peut ensuite être utilisée pour le chiffrement symétrique.

Chiffrement à clef de session Le chiffrement à clef publique est lent (100 à 1000 fois plus lent que le chiffrement à clef privée), c'est pourquoi une technique hybride est généralement utilisée en pratique. L'une des parties génère une clef secrète, appelée clef de session, qu'elle chiffre avec la clef publique de l'autre partie pour la lui envoyer. Ensuite, le chiffrement symétrique est utilisé pour chiffrer le message en utilisant la clef de session échangée.

Signature

Le problème avec le chiffrement est surtout de prouver que le message provient bien de l'expéditeur qui prétend l'avoir envoyé. Chuck pourrait envoyer une requête à Bob en se faisant passer pour Alice : c'est l'attaque MITM que nous avons défini dans la section I.2, partie I.

Ce problème peut être résolu en utilisant la **signature digitale**. C'est un modèle utilisé pour prouver qu'un message provient bien d'une partie donnée. Une manière d'implémenter une signature digitale est d'utiliser le processus inverse au chiffrement asymétrique. Au lieu de chiffrer le message avec la clef publique et le déchiffrer avec la clef privée, la clef privée est utilisée par l'expéditeur pour **signer** le message et le destinataire utilise la clef publique de l'expéditeur pour le déchiffrer. Comme seul l'expéditeur connait la clef privée, le destinataire peut être sûr que le message provient réellement de lui.

En réalité, seule l'empreinte numérique (au lieu de tout le message) est signée par la clef privée. Ainsi, si Alice veut envoyer à Bob un message signé, elle génère l'empreinte du message et la signe avec sa clef privée. Elle envoie le message (en clair) ainsi que l'empreinte signée à Bob. Bob déchiffre l'empreinte signée avec la clef publique d'Alice, calcule l'empreinte à partir du message en clair et vérifie que les deux empreintes sont identiques. Si c'est le cas, Bob est assuré que c'est Alice qui a envoyé le message.

Remarquons ici que la signature digitale ne permet pas de chiffrer le message, ainsi les techniques de chiffrement doivent être utilisées conjointement avec les signatures si la confidentialité est également nécessaire.

Certificats

L'utilisation de la signature digitale permet de prouver qu'un message a été envoyé par une partie donnée, mais comment s'assurer que la clef publique utilisée comme étant celle d'Alice n'est pas réellement celle d'Amanda ? Ce problème peut être résolu en utilisant un **certificat digital**, qui permet d'empaqueter une identité avec la clef publique et qui est signé par une troisième partie appelée **autorité de certification** (CA pour *Certificate Authority*).

Une autorité de certification est un organisme qui vérifie l'identité (dans le sens identité physique réelle) d'une partie et signe la clef publique et l'identité de cette partie avec sa clef

privée. Le destinataire d'un message peut obtenir le certificat digital de l'expéditeur et le vérifier (ou le déchiffrer) avec la clef publique de la CA, préalablement connue par toutes les parties. Cela prouve que le certificat est valide et permet au destinataire d'extraire la clef publique de l'expéditeur pour vérifier sa signature et lui envoyer un message chiffré.

4.3 Délégation

La délégation est un mécanisme de sécurité qui permet à un sujet de déléguer ses permissions et droits à un autre sujet. Quand une entité est approuvée pour la délégation, elle peut représenter l'autre entité et faire appel à des services pour son compte. La délégation est utile si on veut optimiser le nombre d'identités stockées, ou éviter d'avoir un recours systématique à l'autorité de certification.

Il existe deux types de délégation :

- Délégation au niveau de l'**authentification** : Elle est définie si un mécanisme d'authentification fournit une identité différente de l'identité valide de l'utilisateur, à condition que le propriétaire de l'identité effective a déjà autorisé l'autre utilisateur à utiliser sa propre identité.
- Délégation au niveau du **contrôle d'accès** : Elle est réalisée quand un utilisateur délègue certaines de ses permissions à un autre pour accéder à une ressource.

La délégation a été implémentée de plusieurs manières dans la littérature. Nous citons par exemple [Welch03], qui définit la délégation comme étant l'autorisation donnée à un utilisateur d'affecter dynamiquement une nouvelle identité X509 à une entité et de lui déléguer ainsi quelques uns de ses droits. Les utilisateurs créent un certificat de type *proxy* en délivrant un certificat X509 signé par ses propres créances au lieu d'avoir affaire à une autorité de certification. Les certificats proxy peuvent construire des domaines de confiance dynamiquement, en supposant que deux entités qui utilisent des certificats proxy délivrés par la même autorité se font confiance.

[Nikander99] atteste que la délégation veut dire "attribuer à quelqu'un le pouvoir d'agir en tant que représentant". Cela peut faire changer la matrice de contrôle d'accès. La délégation est réalisée grâce à un certificat SPKI (*Simple Public Key Infrastructure*) [1], représenté par 5 champs : C = (I,S,D,A,V)

- **Émetteur** (*I* pour *Issuer*) : l'autorité qui a créé et signé le certificat. Représentée par sa clé publique ou son *hash*.
- **Sujet** (*S pour Subject*) : Partie pour qui le certificat est délivré.

1. **SPKI** : Spécification qui fournit un certificat d'autorisation, associant à une clef publique l'ensemble de droits et privilèges auxquels son propriétaire peut prétendre.

- **Autorité** (*A* pour *Authority*) : Contenu sémantique spécifique à l'application représentant l'autorité.
- **Délégué ?** (*D* pour *Delegated* ?) : Est-ce que cette autorité dans le certificat peut être déléguée à quelqu'un d'autre ?
- **Validité** (*V* pour *Validity*) : Quand est-ce que le certificat est valide ? (période, URL d'un service de vérification en ligne...)

L'émetteur délègue le droit *A* au sujet *S*. Si *S* est une clé publique et si *D* est vrai, alors *S* peut déléguer ce droit à quelqu'un d'autre. La validité de la délégation est limitée par *V*. Le système d'exploitation des nœuds représente la seule source d'autorité primaire du système. La personne ou système qui installe le système d'exploitation pour la première fois a la possibilité de créer des délégations initiales de cette autorité (équivalent à établir un compte administrateur avec un mot de passe).

Conclusion

Tous les mécanismes de sécurité que nous avons présenté visent à assurer une bonne communication entre les différents nœuds d'un système distribué en sécurisant le transport des messages. Cependant, les risques de divulgation d'informations ne se limitent pas aux canaux de communication, mais peuvent parfois survenir au niveau d'un nœud lui-même. L'idée serait donc de garantir la sécurité à la fois entre les nœuds et à l'intérieur d'un même nœud. C'est pour cette raison qu'il est important d'utiliser une représentation explicite pour les systèmes distribués qui puisse implémenter séparément les nœuds et les canaux de communication, tout en répondant aux exigences de modularité, dynamicité et sécurité de ce type de systèmes. Le modèle CBSE semble le plus approprié pour cette tâche.

Chapitre II

Systèmes à base de composants

Nous présentons dans ce chapitre l'ingénierie à base de composants que nous proposons d'utiliser pour faciliter la détection et la correction de la non-interférence dans les systèmes distribués. Nous présentons ensuite deux modèles orientés composants que nous avons utilisé dans nos prototypes, Fractal et SCA.

1 CBSE : Ingénierie logicielle à base de composants

CBSE (*Component-based software engineering*) est une branche de l'ingénierie logicielle permettant de favoriser la séparation des préoccupations selon les fonctionnalités disponibles dans un système logiciel donné, et ce grâce à sa décomposition en **composants**.

Un composant est une unité de composition qui peut être déployée indépendamment et assemblée avec d'autres composants. Grâce à leur modularité, les composants simplifient le développement et la gestion des systèmes distribués.

Plusieurs travaux ont montré le rôle du composant dans l'automatisation de la gestion des systèmes distribués [Abdellatif07, Beisiegel05, Broy98]. Broy et al. définissent les composants comme étant des entités qui doivent (1) encapsuler des données, (2) être implantables dans la plupart des langages de programmation, (3) être hiérarchiquement entrelacés, (4) avoir des interfaces clairement définies et (5) pouvoir être incorporés dans des canevas (*frameworks*).

L'architecture est décrite dans un langage de description d'architecture ADL (*Architecture Description Language*). Le système est ensuite automatiquement déployé sur les hôtes distribués. Chaque composant peut être configuré séparément grâce aux **interfaces de configuration** (ou **attributs**) qui permettent de donner des valeurs aux attributs du composant. Nous distinguons pour chaque composant les **ports serveurs**, qui reçoivent des informations

d'autres composants, des **ports clients**, qui émettent des informations sous forme de messages. De plus, les composants sont faiblement couplés, ce qui veut dire que les connexions entre les différents composants -qu'on appelle **liaisons**- peuvent être établies de différentes manières, indépendamment du code du composant. Une liaison est établie entre un port client et un port serveur.

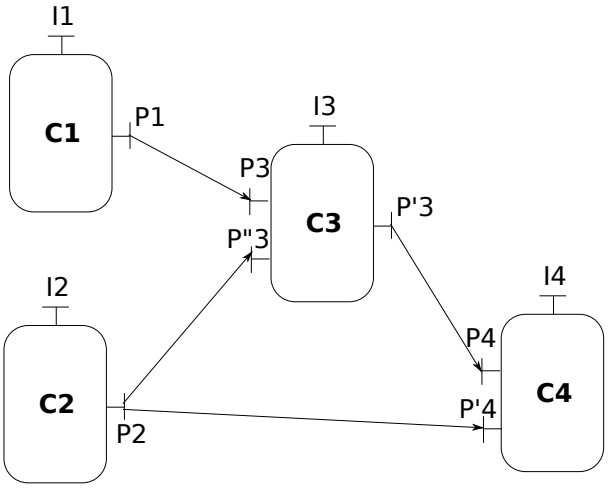


Figure I.II.1 – Exemple d'un système à base de composants

La figure I.II.1 présente un exemple d'un système à base de composants. C_1, C_2, C_3 et C_4 sont des composants. Chaque composant C_i admet deux types d'interfaces : les interfaces de contrôle I_i permettant de configurer le composant et les ports de communication P_i permettant l'envoi de messages d'un composant à un autre. Les ports de communication peuvent être connectés par des liaisons explicites. Dans cet exemple, C_1 est relié à C_3, ce qui veut dire que C_1 peut envoyer des données à C_3 via port P_1, et que C_3 le reçoit via son port P_3.

Il existe deux types de ports : des ports **clients** qui envoient des requêtes et des ports **serveurs** qui les reçoivent. Par exemple, dans le composant C_3, P_3 est un port serveur et $P'3$ est un port client. Par convention, les ports serveurs sont représentés à gauche du composant, et les ports clients à sa droite. Un port client peut être relié à plusieurs ports serveurs, comme par exemple le port P_2 aux ports P'_4 et $P"_3$. Cela implique que la même requête est envoyée à C_3 et C_4.

2 Exemples de modèles à base de composants

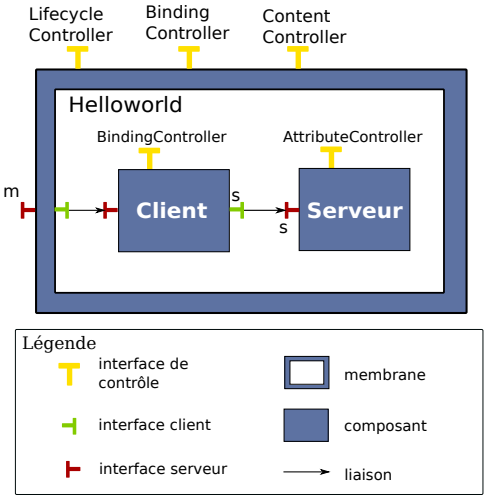
L'ingénierie à base de composants a été utilisée dans plusieurs travaux pour définir un modèle pour la construction des systèmes. Les modèles qui nous intéressent sont ceux qui permettent la séparation nette de l'architecture et de l'implémentation dans des structures différentes.

Plusieurs modèles correspondent à cette description, nous avons choisi deux modèles pour nos prototypes : Fractal et SCA.

2.1 Fractal

Fractal a été réalisé par l'INRIA et l'unité R&D de France Telecom en juin 2002. Fractal est un modèle de composants logiciels modulaire et extensible pour la construction de systèmes répartis hautement adaptables et reconfigurables. Il est utilisé pour implémenter, déployer et reconfigurer les systèmes et applications.

Un composant Fractal est formé de deux parties : un **contrôleur**, également appelé *membrane* et un **contenu**. La figure I.II.2 montre un exemple simple d'un système modélisé avec Fractal.




```
<definition name="helloworld.ClientServerImpl">
   <interface name="m" role="server" signature="helloworld.Main"/>
   <component name="client" definition ="helloworld.ClientImpl"/>
   <component name="serveur" definition="helloworld.ServerImpl"/>
   <binding client="this.m" server="client.m"/>
   <binding client="client.s" server=" serveur.s" />
</definition>
<definition name="helloworld.ClientImpl">
   <interface name="m" role="server" signature="helloworld.Main"/>
   <interface name="s" role="client" signature="helloworld.Service"/>
   <content class="helloworld.ClientImpl" />
</definition>
<definition name="helloworld.ServerImpl">
   <interface name="s" role="server" signature="helloworld.Service"/>
   <content class="helloworld.ClientImpl" />
   <attributes signature="helloworld.ServiceAttributes">
     <attribute name="header" value="->"/>
   </attributes>
   <controller desc="primitive" />
</definition>
```

Figure I.II.2 – Exemple d'une architecture et d'un ADL Fractal

Contenu

Le contenu d'un composant est composé d'un nombre fini d'autres composants, appelés *sous-composants*, qui sont sous le contrôle du contrôleur du composant conteneur. Le modèle Fractal est ainsi récursif. Un composant qui expose son contenu est appelé **composite**. Un composant qui n'expose pas son contenu mais qui a au moins une interface de contrôle est appelé **composant primitif**. Un composant sans interface de contrôle est appelé **composant de base**.

Contrôleur

Le contrôleur d'un composant peut avoir des interfaces *externes*, accessibles de l'extérieur du composant ou *internes*, accessibles uniquement à partir des sous-composants. Une interface **fonctionnelle** est une interface qui correspond à une fonctionnalité fournie (interface *serveur*) ou requise (interface *client*) d'un composant, alors qu'une interface **de contrôle** est une interface serveur qui correspond à un aspect non fonctionnel.

Les interfaces de contrôle sont répertoriées en plusieurs catégories :

- *AttributeController* : Interface responsable de la gestion des attributs d'un composant. Un **attribut** est une propriété configurable d'un composant, généralement de type primitif, et utilisé pour configurer l'état d'un composant.
- *BindingController* : Interface responsable de la gestion des liaisons des interfaces fonctionnelles à d'autres composants.
- *ContentController* : Interface responsable de la gestion des sous-composants d'un composite.
- *LifeCycleController* : Interface responsable de la gestion de l'exécution d'un composant, son démarrage, arrêt, ajout et suppression de sous-composants, liaisons ou attributs, de manière dynamique.

Liaison

Une liaison est un chemin de communication entre des interfaces de composant. Le modèle Fractal distingue entre les liaisons **primitives** et les liaisons **composites**. Une liaison primitive est une liaison entre une interface client et une interface serveur d'un même espace d'adressage. Une liaison composite est un chemin de communication entre un nombre aléatoire d'interfaces de composants et de types de langages. Ces liaisons sont représentées par un ensemble de liaisons primitives et de composants de communication (appelés aussi *connecteurs*).

2.2 SCA : Architecture de services à base de composants

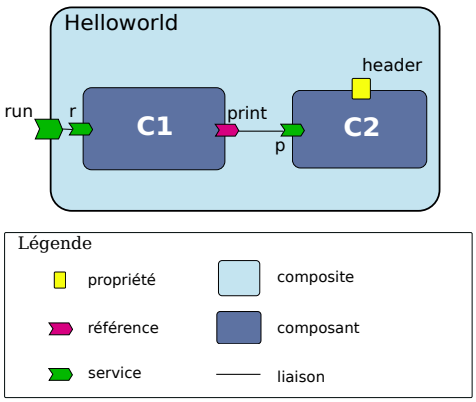
SCA (**S**ervice **C**omponent **A**rchitecture) [1] est le produit d'un effort entre plusieurs entreprises telles que BEA, IBM, Oracle et Sun, devenue un standard OASIS. Elle offre un ensemble de spécifications visant à simplifier la construction de systèmes orientés service, indépendamment de leurs implémentations et en utilisant des composants qui implémentent une logique métier. Le but de SCA est de simplifier la création, l'implémentation et le déploiement de ser-

1. **SCA** : http ://www.osoa.org/display/Main/Service+Component+Architecture+Home

vices dans l'architecture SOA (*Service-Oriented Architecture*) en faisant abstraction des détails des communications sous-jacentes.

Modèle de composition

L'entité principale d'un modèle SCA est le composant, qui implémente une certaine logique métier et qui peut être assemblé en différentes configurations. Un composant est une instance d'une implémentation configurée ; il offre ses *services*, requiert des *références* d'autres composants et définit des *propriétés*. Il spécifie la manière dont il communique avec un autre composant via des *liaisons*. Le modèle de composition SCA aide à assembler récursivement des composants à granularité fine et étroitement couplés pour former d'autres composants à grosse granularité appelés des *composites*. La figure I.II.3 représente un exemple d'un système à base de composants modélisé avec les notations graphiques du standard SCA, ainsi qu'un extrait du fichier *composite* qui décrit l'architecture du système grâce au langage ADL de SCA.




```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           name="Helloworld">
 <service name="run" promote="C1/r"/>
 <component name="C1">
   <implementation.java class ="helloworld.Client"/>
   <service name="r">
     <interface.java interface="java.lang.Runnable"/>
   </service>
   <reference name="print" target="C2/p">
     <interface.java interface="helloworld.PrintService"/>
   </reference>
 </component>
 <component name ="C2">
   <implementation.java class="helloworld.Server"/>
   <service name="p">
     <interface.java interface="helloworld.PrintService"/>
   </service>
   <property name="header">  -->  </property>
 </component>
</composite>
```

Figure I.II.3 – Exemple d'une architecture SCA

Implémentations

Les éléments SCA définis ci-dessus, c.-à-d. les références, services, propriétés et liaisons, doivent être utilisés de manière transparente à l'utilisateur. Plusieurs implémentations sont proposées par les fournisseurs et les projets open-source. Nous en citons deux : Tuscany [2] et Frascati [3].

Tuscany est considérée comme étant l'implémentation de référence de SCA. Elle offre une infrastructure permettant de développer et d'exécuter facilement des applications SCA, en

2. **Tuscany** : http ://tuscany.apache.org/
3. **Frascati** : frascati.ow2.org/

fournissant un support pour différentes technologies (tel que web20 et OSGi) et plusieurs types d'implémentations (tel que Spring, BPEL et Java). Elle cible à réduire le coût de développement en utilisant des protocoles configurables séparés de la logique métier.

Frascati [Seinturier09] est une plateforme pour les applications SCA qui étend le modèle de composants avec des capacités réflexives permettant l'intégration dynamique des propriétés non-fonctionnelles des composants grâce à des *intercepteurs*. Le but principal du modèle Frascati est de résoudre les problèmes de gestion et de configuration de la plateforme SOA.

Les Politiques (*Policies*)

Les **politiques** [Beisiegel07] sont utilisées dans SCA pour représenter les aspects non-fonctionnels de l'application, tel que le recueil de données (*logging*), la supervision (*monitoring*) et la sécurité. Le modèle SCA les représente en termes d'**intentions** (*intents*) et de **paramètres de politique** (*policy sets*).

Les *intents* décrivent les besoins abstraits des composants, services et références. Les *policy sets* sont utilisés pour implanter les caractéristiques énoncées dans les *intents* en appliquent des politiques particulières à un composant ou à une liaison d'un service ou d'une référence, tout en spécifiant les détails techniques.

Les politiques qui nous concernent sont les politiques de sécurité. Les mécanismes de sécurité disponibles avec le modèle SCA sont principalement le contrôle d'accès, l'authentification, la confidentialité et l'intégrité. Un *intent* spécifie les besoins de sécurité de manière abstraite, tel que par exemple la *confidentialité*, sans préciser comment la fournir. Cet *intent* est associé à un artéfact SCA (référence ou service, par exemple) et relié à une WS-Policy à travers les *policy sets*, pour spécifier explicitement les besoins techniques de la politique, comme par exemple la méthode de chiffrement requise pour assurer la confidentialité.

Conclusion

Nous avons comme objectif dans notre travail d'utiliser le paradigme orientés composants pour faciliter l'application et la vérification d'une propriété de contrôle de flux d'information, appelée la non-interférence. Le chapitre suivant définit cette propriété ainsi que les mécanismes permettant de l'appliquer sur les systèmes réels.

Chapitre III

La non-interférence

Nous présentons dans ce chapitre la propriété de sécurité ciblée par notre travail, la **non-interférence**, une propriété qui a suscité l'intérêt de plusieurs travaux de recherche. Nous présentons d'abord une définition formelle de cette propriété qui nous a paru appropriée pour les types de systèmes que nous ciblons. Nous expliquons ensuite les formats d'étiquettes de sécurité utilisées pour appliquer les niveaux de sécurité aux différents éléments manipulés par le système.

1 La non-interférence

La non-interférence est un modèle pour le contrôle de flux d'information (CFI) définie par Goguen et Meseguer [Goguen82], qui assure que les données sensibles n'affectent pas le comportement publiquement visible du système. Cette propriété de sécurité permet le suivi de l'information dans le système, et l'application de la confidentialité et de l'intégrité de bout en bout. Nous présentons dans ce qui suit une définition formelle de cette propriété.

1.1 Définition formelle de la non-interférence

Il existe plusieurs définitions de la non-interférence dans l'état de l'art [Goguen82, Rushby92, Malecha10, Myers06]. Pour notre contexte, la définition de [Malecha10] nous paraît la plus adéquate, car elle est applicable aussi bien pour un programme que pour un ensemble de composants communiquant via le réseau. Dans ce travail, la non-interférence se ramène à un problème d'**indiscernabilité** (*indistinguishability*) entre plusieurs états de la mémoire, le réseau étant abstrait à une mémoire partagée.

Considérons le vocabulaire suivant :

- s : programme
- L : ensemble de niveaux de sécurité dans le programme
- $\subseteq_L$ un ordre partiel sur L, il définit les informations qui peuvent circuler entre les niveaux de sécurité. Une information est autorisée à circuler d'un niveau p à un niveau q si et seulement si $p \subseteq_L q$. On dit alors que le niveau q est *au moins aussi restrictif que* le niveau p. $\bot \in L$ représente le niveau de sécurité le plus bas : $\bot \subseteq_L p, \forall p \in L$
- $\phi = (L, \subseteq_L)$: la politique de sécurité
- $\bigcup_L$: opérateur de jointure. Pour les niveaux de sécurité p et q, il existe $p \bigcup_L q \in L$ qui a pour limite supérieure à la fois p et q.
- $\sigma[x \mapsto v]$: mémoire. Représente une configuration associant une variable x à une valeur v.
- Γ : environnement de variables, représente un sous-ensemble de variables, dont un observateur à un niveau de sécurité $o \in L$ peut observer les valeurs.
  - $\Gamma \vdash \sigma_1 \approx_o \sigma_2$: pour toutes les variables x que le niveau de sécurité o peut observer, on a : $\sigma_1(x) = \sigma_2(x)$. On dit que σ_1 et σ_2 sont **indiscernables** par rapport à o.
  - On dit aussi que "Γ protège x au niveau o" si x n'est observable à aucun niveau de sécurité moins restrictif que o.
- $\phi \vdash s, \sigma \rightarrow^* v, \sigma'$: fermeture transitive de la sémantique opérationnelle à petits pas, indiquant que pour la politique de sécurité ϕ et avec la configuration σ, le programme s est évalué à la valeur v et à la mémoire σ'.

Un programme s satisfait la propriété de non-interférence pour un niveau de sécurité o sous l'environnement Γ si, pour toutes les politiques de sécurité ϕ, pour toutes les configurations σ, pour toutes les variables h telles que Γ protège h à un niveau o' avec $o' \not\subseteq_L o$ et pour toutes les valeurs $v1$ et $v2$ du même type :

Théorème 1. ***si*** $\phi \vdash s, \sigma[h \mapsto v_1] \rightarrow^* v_1', \sigma_1'$ ***et*** $\phi \vdash s, \sigma[h \mapsto v_2] \rightarrow^* v_2', \sigma_2'$
alors $\Gamma \vdash \sigma_1' \approx_o \sigma_2'$ ***et*** $v_1' = v_2'$

Ainsi, si un attaquant peut observer des données jusqu'à un niveau de sécurité o, alors une modification d'une variable de niveau de sécurité plus haut est indiscernable pour cet attaquant.

Cette définition induit en fait que les résultats (ou sorties) observables à un niveau o doivent être indépendants des entrées à des niveaux plus restrictifs que o.

Nous pouvons généraliser cette définition au cas des systèmes répartis. Un système réparti est constitué d'un ensemble de processus distribués sur des sites qui communiquent grâce à un réseau de communication. Chaque processus p est caractérisé par un ensemble d'entrées I et de sorties O auxquels sont associés des niveaux de sécurité $l \in L$. Ainsi, pour chaque processus p, la modification des valeurs des entrées ayant un niveau de sécurité o' doit être indiscernable

pour tout attaquant capable d'observer des sorties à un niveau o, si $o' \not\sqsubseteq_L o$, ce qui implique que les sorties d'un processus doivent être indépendantes de l'ensemble de ses entrées dont le niveau de sécurité est plus restrictif. Dans le cas des systèmes distribués à base de composants, un processus est considéré comme étant une instance de composant.

1.2 Exemples d'interférence dans le code

La violation de la propriété de non-interférence peut se manifester de plusieurs manières dans le code d'un composant. Nous présentons trois exemples illustrant des fuites de sécurité dans un programme. Supposons dans ce qui suit que les variables h, h_1 et h_2 représentent des données sensibles, alors que les variables l, l_1 et l_2 représentent des données publiques.

Flux explicite

Considérons une instruction d'affectation simple : $l := h$. L'information circule d'une entrée privée h vers une sortie publique l. Ceci représente le cas d'un flux **explicite** illégal.

Flux implicite

Pour le deuxième exemple, considérons l'instruction suivante :

$$\text{si } h \text{ alors } l := vrai \text{ sinon } l := faux\,;$$

La valeur de la sortie publique l révèle celle de l'entrée privée h. La variable publique l a été modifiée dans un contexte de niveau de sécurité *High*, ce qui représente un exemple d'un flux **implicite** illégal. Les flux implicites circulent dans des canaux cachés (*covert channels*) et ne peuvent être gérés correctement par un mécanisme purement dynamique tel que le contrôle d'accès, apparenté à la classe EM (pour *Execution Monitoring*) définie par [Schneider00].

Référencement

Le troisième exemple concerne le cas particulier des langages orientés objet. Le mécanisme de référencement (*aliasing*) des objets peut créer un problème d'interférence. Un même objet peut être référencé par plusieurs variables de niveaux de sécurité différents. Considérons l'exemple suivant inspiré de [Amtoft06] : soit la variable privée h_1 qui référence un objet o contenant un attribut i public (initialisé à 0), et soient deux variables publiques l_1 et l_2. Soit l'instruction suivante :

si $(h_2 > 0)$ **alors** {
    $h_1 := l_1$;
} **sinon** {
    $h_1 := l_2$;
}
$h_1.i := 40$;

A première vue, ces instructions ne contiennent pas de flux explicites illégaux, car la circulation de l'information se fait des données publiques vers les données privées, ni de flux implicites, car seule une variable privée (h_1) a été modifiée dans un contexte privé. Cependant, comme h_1, l_1 et l_2 sont des objets, alors, suite à ces instructions, si h_2>0 alors les variables h_1 et l_1 référencent le même objet, sinon ce sont h_1 et l_2 qui référencent le même objet. Ainsi, puisque l'attribut i est public, alors en observant sa valeur dans l_1 et l_2 on peut déduire le signe de h_2 : si $l_1.i = 40$, alors h_1 et l_1 référencent le même objet, donc h_2 est positif, et si $l_2.i = 40$ alors h_1 et l_2 référencent le même objet, donc h_2 est négatif. Ainsi, pour empêcher ces problèmes dus au référencement, il faut s'assurer que **les attributs publics d'un objet (*i* dans notre exemple) ne soient pas modifiés à partir d'une référence privée (h_1)**. Partant du même principe, un appel de méthode $h_1.m()$ ne doit pas modifier des variables publiques si h_1 est privé.

Dans ce travail, nous nous restreignons à ces trois types d'interférences. Nous ne couvrons pas les autres aspects avancés de canaux cachés, tels que les canaux temporels et l'exploitation des ressources partagées.

2 Étiquettes de sécurité

Les **étiquettes de sécurité** (*labels*) sont utilisées pour annoter les variables du programme dans le but de contrôler le flux d'information dans les programmes. L'étiquette d'une variable contrôle la manière dont la donnée stockée dans cette variable peut être utilisée. La clé pour protéger la confidentialité et l'intégrité est d'assurer que, quand la donnée circule dans le système, ses étiquettes deviennent de plus en plus restrictives. Si le contenu d'une variable affecte celui d'une autre variable, c'est qu'il existe un flux d'information de la première vers la seconde : ainsi, l'étiquette de la seconde variable doit être au moins aussi restrictive que celle de la première.

Il existe plusieurs modèles d'étiquettes dans la littérature. Nous allons en présenter trois, que nous retrouvons respectivement dans [Myers00, Krohn07, Khair98]. Le modèle d'étiquettes et son attribution aux différentes parties du système définit la politique de sécurité du système.

2.1 DLM : Modèle d'étiquettes décentralisé

Cette section décrit le modèle d'étiquettes décentralisé (DLM : *Decentralized Label Model*) [Myers00]. Ce modèle est dit "décentralisé" car les politiques de sécurité ne sont pas définies par une autorité centrale, mais contrôlée par les différents participants au système. Le système doit ensuite se comporter de manière à respecter ces politiques de sécurité. Cette particularité est possible grâce à la notion de "propriété" des étiquettes utilisées pour annoter les données. Cette notion autorise chaque participant à *rétrograder* le niveau de sécurité de ses données, mais pas celles des autres.

Les entités principales de ce modèle sont les **autorités** (*principals*), dont on veut protéger les informations secrètes, et les **étiquettes** (*labels*), qui représentent la manière dont les autorités expriment leurs contraintes.

Autorités

Les autorités représentent les entités qui possèdent, modifient et publient les informations. Ils représentent les utilisateurs, groupes ou rôles. Un processus a le droit d'agir au nom d'un ensemble d'autorités.

Quelques autorités ont le droit d'agir pour d'autres. Quand une autorité A peut agir pour une autre autorité *A'*, A possède tous les pouvoir et privilèges potentiels de *A'*. La relation *agit pour* est réflexive et transitive, et permet de définir une hiérarchie ou un ordre partiel entre les autorités.

La relation *A agit pour A'* est notée $A \succeq A'$. Elle permet la délégation de tous les pouvoirs d'une autorité (A') à une autre (A).

La gestion de la hiérarchie des autorités peut également être faite de manière décentralisée. La relation $A \succeq A'$ peut être ajoutée à la hiérarchie tant que le processus qui se charge de l'ajout a suffisamment de privilèges pour agir pour l'autorité *A'*. Aucun privilège n'est nécessaire concernant A car cette relation donne plus de pouvoirs à A.

Étiquettes

Les autorités expriment leurs contraintes de sécurité en utilisant des étiquettes pour annoter les programmes et les données. Une étiquette est composée de deux majeures parties : une

étiquette de confidentialité et une étiquette d'intégrité. Chacune de ces étiquettes contient un ensemble d'éléments qui expriment des besoin de sécurité définis par des autorités. Un élément d'étiquette a deux parties : un **propriétaire** et un ensemble d'autorités, qui représentent des **lecteurs** dans une étiquette de confidentialité et des **écrivains** dans une étiquette d'intégrité.

L'objectif d'un élément d'étiquette est de protéger les données privées du propriétaire de l'élément. Il est alors appelé **politique d'utilisation de la donnée**. Ainsi, une donnée est étiquetée avec un certain nombre de *politiques* énoncées par les autorités propriétaires de la donnée.

Étiquettes de confidentialité Une étiquette de confidentialité exprime le niveau de secret désiré par le propriétaire d'une donnée en citant la liste les **lecteurs** potentiels. Les lecteurs sont les autorités auxquelles cet élément permet de consulter la donnée. Ainsi, le propriétaire est la source de la donnée et les lecteurs sont ses destinataires possibles. Les autorités qui ne sont pas listées comme lecteurs n'ont pas le droit de consulter les données.

Un exemple d'étiquette est :

$$L_C = \{o_1 \to r_1, r_2; o_2 \to r_2, r_3\} \tag{III.1}$$

Ici, o_1, o_2, r_1 et r_2 sont des autorités. Le point-virgule sépare deux politiques (éléments) de l'étiquette L_C. Les propriétaires de ces politiques sont o_1 et o_2 et les lecteurs des politiques sont respectivement $\{r_1, r_2\}$ et $\{r_2, r_3\}$. La signification d'une étiquette est que chaque politique dans l'étiquette doit être respectée pendant que la donnée circule dans le système, de manière à ce que chaque information étiquetée ne soit divulguée que suite à un consensus entre TOUTES les politiques. Un utilisateur peut lire les données seulement si *l'autorité représentant cet utilisateur peut agir pour un lecteur de chaque politique dans l'étiquette.* Ainsi, seuls les utilisateurs dont les autorités peuvent agir pour r_2 peuvent lire les données étiquetées par L. La même autorité peut être le propriétaire de plusieurs politiques ; les politiques restent renforcées indépendamment l'une de l'autre dans ce cas.

Étiquettes d'intégrité Les politiques d'intégrité sont duales aux politiques de confidentialité. Comme les politiques de confidentialité protègent contre les données lues de manière impropre, même si elles traversent ou sont utilisées par des programmes qui ne sont pas de confiance, les politiques d'intégrité protègent les données contre une modification inappropriée. Une étiquette d'intégrité garde la trace de toutes les sources qui ont affecté sa valeur, même si ces sources l'affectent indirectement. Elles empêchent les données non fiables d'avoir un effet sur les données de confiance stockées.

La structure d'une politique d'intégrité est identique à celle d'une politique de confidentialité. Elle a un propriétaire, l'autorité pour laquelle la politique est appliquée, et un ensemble d'**écrivains**, des autorités qui sont autorisées à modifier la donnée. Une étiquette d'intégrité peut contenir un ensemble de politiques d'intégrité avec différents propriétaires. Un exemple d'étiquette est : $L_I = \{o_1 \leftarrow w_1, w_2; o_2 \leftarrow w_2, w_3\}$ avec o_1 et o_2 les propriétaires de la politique, et w_1, w_2 et w_3 les écrivains.

Intuitivement, une politique d'intégrité est une garantie de qualité. Une politique $\{o \leftarrow w_1, w_2\}$ est une garantie fournie par l'autorité o que seuls w_1 et w_2 sont capables de modifier la valeur de la donnée. L'étiquette d'intégrité la plus restrictive est celle qui ne contient aucune politique : $\{\}$. C'est l'étiquette qui ne fournit aucune garantie sur le contenu de la valeur, et peut être utilisée comme donnée d'entrée uniquement quand le destinataire n'impose aucune exigence d'intégrité. Quand une étiquette contient plusieurs politiques d'intégrité, ces politiques sont des garanties indépendantes de qualité de la part des propriétaires de ces politiques. En utilisant une étiquette d'intégrité, une variable peut être protégée contre des modifications impropres.

Par exemple, supposons qu'une variable a une politique unique $\{o \leftarrow w_1, w_2\}$. Une valeur étiquetée $\{o \leftarrow w_1\}$ peut être introduite dans cette variable car cette valeur a été affectée uniquement par l'autorité w_1 et l'étiquette de la variable autorise w_1 à la modifier. Si la valeur était étiquetée $\{o \leftarrow w_1, w_3\}$, l'écriture ne serait pas autorisée, car la valeur a été affectée par w_3, une autorité qui n'est pas mentionnée comme étant un écrivain légal dans l'étiquette de la variable. Cela serait uniquement permis si $w_3 \succeq w_2$. Enfin, considérons une valeur étiquetée $\{o \leftarrow w_1; o' \leftarrow w_3\}$. Dans ce cas, l'écriture est permise, car la première politique atteste que o croit que seul w_1 a altéré la valeur. Le fait que la seconde politique existe de la part de o' n'affecte pas la légalité de l'écriture dans la variable ; c'est une garantie supplémentaire de qualité.

2.2 Modèle d'étiquettes à base de jetons

Le modèle suivant, inspiré de [Krohn07, Zeldovich06], représente les étiquettes sous la forme d'un ensemble de jetons (*tags*). Un jeton est un terme opaque qui, sorti de son contexte, n'a pas de signification précise, mais qui est attribué aux données pour leur associer une certaine catégorie de confidentialité ou d'intégrité. Une étiquette l appartenant à un ensemble L de niveaux de sécurité, contient ainsi deux ensembles : S (pour ***S**ecrecy*) qui représente le niveau de confidentialité et I (pour ***I**ntegrity*) qui représente le niveau d'intégrité. On note alors $l = \{S; I\}$. Dans ce qui suit, nous nous basons sur ce modèle pour illustrer les relations d'ordre entre les étiquettes.

Confidentialité Associer un jeton j de confidentialité ($j \in S$) à une donnée implique que cette donnée contient une information privée de niveau de confidentialité j. Pour révéler le contenu d'une donnée, le système doit obtenir l'accord pour chacun des jetons de confidentialité qui ornent cette donnée.

Intégrité Le niveau d'intégrité d'une donnée représente une garantie de l'authenticité de l'information dans cette donnée. Il permet au destinataire de s'assurer que la donnée qui lui a été envoyée n'a pas été modifiée par des parties non fiables. Attribuer un jeton i d'intégrité ($i \in I$) à une donnée représente une garantie supplémentaire pour cette donnée.

Relation d'ordre L'ensemble des étiquettes dans un système est régi par la relation d'ordre partielle "peut circuler vers" (*can flow to*), représentée par le symbole $\subseteq$. Cette relation ordonne les étiquettes de la moins restrictive vers la plus restrictive.

La relation "peut circuler vers" peut être décomposée en deux sous-relations :

- $\subseteq_C$: Cette relation ordonne les niveaux de confidentialité
- $\subseteq_I$: Cette relation ordonne les niveaux d'intégrité

Définition 1 (Relation d'ordre)**.** Soient deux étiquettes $l_1 = \{S_1; I_1\}$ et $l_2 = \{S_2; I_2\}$. On dit que :

$$l_1 \subseteq l_2 \text{ si et seulement si } S_1 \subseteq_C S_2 \text{ et } I_1 \subseteq_I I_2$$

Définition 2 (Confidentialité)**.** Soient S_1 et S_2 deux niveaux de confidentialité, formés chacun d'un ensemble de jetons.

$$S_1 \subseteq_C S_2 \text{ si et seulement si } \forall j_1 \in S_1, \exists j_2 \in S_2 \text{ tel que } j_1 = j_2$$

Définition 3 (Intégrité)**.** Soient I_1 et I_2 deux niveaux d'intégrité, formés chacun d'un ensemble de jetons.

$$I_1 \subseteq_I I_2 \text{ si et seulement si } \forall i_2 \in I_2, \exists i_1 \in I_1 \text{ tel que } i_1 = i_2$$

Théorème 2 (Réunion)**.** *Soient deux étiquettes* $l_1 = \{S_1; I_1\}$ *et* $l_2 = \{S_2; I_2\}$*. On définit* $l = \{S; I\}$*, la réunion de* l_1 *et* l_2*, par :*

$$l = l_1 \bigcup l_2 \Leftrightarrow S = S_1 \bigcup S_2 \text{ et } I = I_1 \bigcap I_2$$

Corollaire. si l_1 et l_2 sont des étiquettes de L, alors $l_1 \bigcup l_2 \in L$, tel que :

$$\forall l \in L, \text{ si } l \subseteq l_1 \text{ et } l \subseteq l_2 \text{ alors } l \subseteq l_1 \bigcup l_2$$

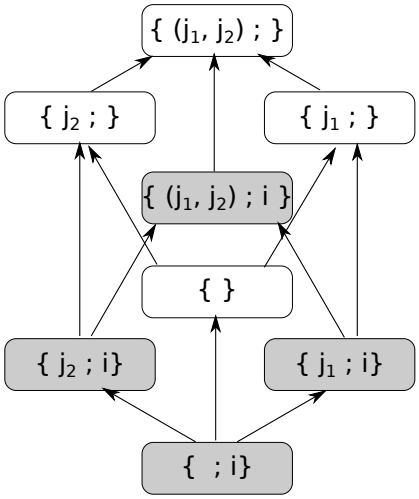


Figure I.III.1 – Exemple d'un treillis de sécurité [Zeldovich06]

Théorème 3 (Transitivité)**.** *Soient deux étiquettes* l_1 *et* l_2*.*

$$Si\ l_1 \subseteq l_2\ et\ l_2 \subseteq l_3\ alors\ l_1 \subseteq l_3$$

Théorème 4 (Réflexivité)**.** *Soient deux étiquettes* l_1 *et* l_2*.*

$$l_1 \subseteq l_2\ et\ l_2 \subseteq l_1 \Leftrightarrow l_1 = l_2$$

La relation d'ordre $\subseteq$ permet de définir un **treillis de sécurité**, qui ordonne l'ensemble des étiquettes d'un système de la moins restrictive (en bas du treillis) vers la plus restrictive (en haut du treillis). Le treillis nous permet ainsi de voir l'acheminement autorisé du flux d'information : une information ne peut circuler que vers des cibles plus restrictives, dont les étiquettes sont positionnées plus haut dans le treillis de sécurité.

La figure I.III.1 (extraite de [Zeldovich08]) représente un exemple de treillis, où deux niveaux de confidentialité (j_1 et j_2) et un niveau d'intégrité (i) sont définis. Les cellules grisées représentent les étiquettes qui incluent le niveau d'intégrité i. Les flèches allant d'une étiquette l_1 vers une étiquette l_2 indiquent qu'un flux d'information allant d'une donnée étiquetée l_1 vers une donnée étiquetée l_2 est autorisé.

La relation $\subseteq$ est une relation **réflexive**. Sur la figure I.III.1, les flèches allant d'une étiquette vers elle-même ne sont pas représentées.

Si deux étiquettes l_1 et l_2 ne sont pas reliées par une flèche dans le treillis et aucun chemin ne permet d'aller de l'une à l'autre (comme par exemple les étiquettes $\{j_1;\}$ et $\{j_2; i\}$ dans la figure I.III.1), on dit qu'elles sont **incomparables**, et on note :

$$l_1 \not\subseteq l_2 \text{ et } l_2 \not\subseteq l_1$$

2.3 Modèle d'étiquettes distribué

Dans [Khair98], les étiquettes de sécurité contiennent deux parties :

* **Niveau** Les niveaux de sécurité attribués à chaque utilisateur, donnée et site sont définis après une première étude qui résulte en une classification des droits des utilisateurs, la sensibilité des données et le degré de confiance des sites.
* **Catégorie**
  - La catégorie d'un utilisateur dépend de sa position dans la hiérarchie des rôles (URH : *User Role Hierarchy*). La catégorie la plus haute qui correspond à la racine de l'arbre URH contient toutes les sous-catégories et rôles définis dans l'application.
  - La catégorie des données dépend de leur utilisation et correspond aux besoins de certaines catégories d'utilisateurs.
  - La catégorie d'un site dépend de son utilisation (quelles catégories d'utilisateurs l'utilisent et quelles catégories de données a-t-il besoin d'utiliser), et de son type (autonome, connecté à internet, mobile, stationnaire...).

Affectation des étiquettes de sécurité aux utilisateurs L'affectation des étiquettes à chaque rôle se fait en commençant à partir de la racine et allant vers le dernier niveau avant les feuilles de l'arbre URH. Une *catégorie* est affectée au rôle de l'utilisateur, puis un *niveau* leur est attribué selon la confiance donnée à cet utilisateur ainsi qu'à son niveau de responsabilité.

Affectation des étiquettes de sécurité aux objets On commence d'abord par attribuer les ensembles de données aux feuilles de l'arbre URH correspondant aux données qui ont besoin d'être accédées par les rôles. Le concepteur utilise les exigences d'accès à l'information comme guide pour assurer que les privilèges adéquats sont donnés. La procédure est répétée jusqu'à atteindre la racine. Les étapes sont les suivantes :

- Attribuer les ensemble de données en commençant par les rôles sous une catégorie d'utilisateurs.
- Déplacer les données partagées par tous les rôles d'utilisateurs vers leur catégorie d'utilisateur commune.
- Déplacer les données partagées par toutes les catégories d'utilisateurs vers leur super-catégorie, et répéter l'opération jusqu'à atteindre la racine.
- Attribuer une étiquette aux données contenant la catégorie du nœud ou de la feuille déjà attribués dans l'arbre URH, et leur donner comme niveau de sécurité le plus bas niveau d'habilitation (*clearance*) des rôles d'utilisateurs contenus dans la catégorie.

– Répéter ce processus pour toutes les données de chaque rôle et de chaque catégorie d'utilisateur.

Dans le but de vérifier la sécurité au niveau de cette étape, une classification de la sensibilité des données peut être réalisée. Cette classification dépend du contenu et du contexte de la donnée.

Dans le cas où les niveaux diffèrent, l'architecte de la sécurité doit vérifier et décider quel niveau affecter. En complétant cette étape, le concepteur obtient une identification initiale des privilèges de chaque nœud URH et une étiquette initiale affectée à chaque ensemble de données.

Affectation des étiquettes de sécurité aux sites L'étiquette de sécurité de chaque site est dérivée de la fonction des paramètres physiques et opérationnels du site, ainsi que du type de la connexion et sa catégorie de mobilité.

La catégorie d'un site dépend de son utilisation, c.-à-d. la catégorie des utilisateurs ayant un accès direct au site. Le niveau de sécurité du site dépend de plusieurs paramètres relatifs au site, tel que la vulnérabilité du système d'exploitation et du système de gestion de la base de données, de l'environnement physique et de la nature du site, etc. La vulnérabilité du système d'exploitation et du système de gestion de la base de données peut être évaluée en utilisant des critères et méthodes d'évaluation nationales ou internationales de la sécurité. La mobilité du site est également une caractéristique très significative, puisque les vulnérabilités diffèrent dans le cas d'un système mobile et d'un système stationnaire.

Conclusion

La non-interférence est une propriété de sécurité stricte qui permet de contrôler le flux d'information et d'interdire ainsi toute divulgation d'information directe ou indirecte susceptible de menacer la confidentialité ou l'intégrité des données. Son application sur les systèmes distribués est d'autant plus ardue que ces systèmes ont la particularité d'être dynamiques, indépendants et de coexister dans un environnement peu sûr. Il existe un certain nombre de difficultés à cette tâche, que nous énumérons dans la partie suivante.

Chapitre IV

Problématique : Application de la non-interférence aux systèmes distribués dynamiques

Dans ce chapitre, nous montrons les difficultés qui se posent pour assurer le contrôle de flux dans les systèmes distribués et pour appliquer ce contrôle dans les systèmes réels.

1 Contrôle de flux d'information et systèmes distribués

Les approches de sécurité standard pour les systèmes distribués, que nous avons citées ci-dessus, sont basées essentiellement sur les politiques de contrôle d'accès qui contrôlent l'exécution des actions sur les objets individuels [Hutter06]. Cependant, les politiques de contrôle d'accès souffrent du problème de **Cheval de Troie** ou d'autres fuites d'information utilisant des canaux cachés. Cela est dû au fait qu'aucun contrôle n'est fait sur l'utilisation d'une donnée une fois délivrée au service autorisé. Le calcul inapproprié de l'information confidentielle et les appels à des services tiers peut divulguer des secrets à d'autres entités qui ne sont pas autorisées à lire cette information. Le contrôle du flux d'information obligatoire peut résoudre certains de ces problèmes en étiquetant chaque information et service avec un niveau de sécurité déterminé.

Plusieurs travaux ont traité la problématique de la non-interférence qui a été appliquée aux systèmes centralisés et distribués. Les solutions de contrôle de flux d'information pour les systèmes centralisés sont surtout basées sur l'analyse de flux, réalisée avec des langages **typés sécurité**, qui entremêlent la politique de sécurité avec le code fonctionnel de l'application, et permettent ainsi de créer des systèmes non-interférents par construction. Ces langages ont l'avantage de traiter le flux d'information à un niveau de granularité très fin, mais souffrent de

certains inconvénients : les contraintes de sécurité et le comportement fonctionnel du système ne sont pas séparés, ce qui implique que, d'une part, le développeur doit être suffisamment calé en sécurité pour appliquer ces contraintes lui-même sur le code, et d'autre part, qu'il doit subir une phase d'apprentissage assez lourde pour prendre en main le nouveau langage.

Les solutions distribuées pour la vérification de la non-interférence sont composées de deux catégories : la première catégorie est composée de solutions qui traitent le flux d'information à un niveau de granularité assez grossier, ce qui risque d'autoriser certaines fuites d'information dues aux flux implicites, par exemple. Les solutions de la deuxième catégorie travaillent certes à un niveau plus fin, mais proposent de répartir le système après l'application des contraintes de sécurité, en le divisant en modules non-interférents mutuellement. Le problème c'est que les systèmes créés par ces solutions sont distribués selon les contraintes de sécurité, pas selon les contraintes fonctionnelles.

Ces solutions de contrôle de flux d'information réalisent un contrôle statique qui se fait à la compilation. En effet, le but principal de notre travail est de réaliser des systèmes distribués à base de composants sécurisés **par construction**. Cependant, une fois le système compilé et prêt à l'exécution, il est important de réfléchir au maintien de cette sécurité pendant l'exécution, car les applications que nous réalisons sont des applications distribuées, donc hautement dynamiques. Les composants sont en continuelle évolution et rien ne garantit que la structure du système reste figée dans le temps : en effet, l'ajout d'un nouveau composant ou le remplacement d'un composant suite à une panne sont des opérations tout à fait usuelles. Il est donc important que ces évolutions soient prises en considération dans l'étude de la non-interférence.

2 Implémentation de la non-interférence dans les systèmes réels

Même si le domaine de la non-interférence est largement étudié, surtout récemment, son application aux systèmes à grande échelle reste ardu. En effet, selon [Zdancewic04], plusieurs problèmes font que le contrôle des flux d'information n'est pas appliqué :

- La sécurité est en général définie relativement à un certain niveau d'abstraction, et il existe toujours des attaques qui violent cette abstraction.
- Il est difficile d'implémenter un système "pratique" pour le renforcement des politiques de flux d'information. Ce système doit faciliter l'intégration avec les infrastructures existantes.
- La non-interférence est une propriété très restrictive qui n'est pas toujours voulue par les concepteurs du système.

C'est pour ces raisons qu'une solution pour la non-interférence utilisable dans les systèmes réels doit être **flexible**, **intégrable** et **facilement applicable**, en particulier pour les personnes peu expertes en sécurité. Elle doit également pouvoir assurer d'autres propriétés de sécurité avec la non-interférence.

En effet, l'application de la non-interférence sur les systèmes complexes est une tâche ardue, car l'information peut prendre plusieurs formes et circuler sur plusieurs types de canaux, ce qui rend son suivi difficile. Le contrôle du flux d'information doit également être mélangé à d'autres types de mécanismes de sécurité, car les systèmes distribués en général font face à plusieurs menaces extérieures dues à l'interaction de leurs entités dans des réseaux peu fiables. L'utilisation des systèmes à base de composants peut donc s'avérer judicieuse pour faciliter la détection des interférences, grâce à leur modularité et aux interfaces distinctes reliant leurs composants. Cependant, ce type de systèmes n'a pas été exploité pour réaliser cette tâche dans l'état de l'art.

Conclusion

Dans cette partie, nous avons introduit le contexte de notre travail, qui se divise principalement en deux volets : la sécurité des systèmes distribués, et l'étude de la propriété de non-interférence. La problématique principale consiste à appliquer cette propriété sur ces types de systèmes.

Nous présentons dans ce qui suit l'état de l'art sur le sujet. Nous l'avons divisé en deux catégories : Les solutions existantes pour assurer la sécurité des systèmes répartis, et les outils, modèles et langages utilisés pour le contrôle de flux d'information.

Deuxième partie

Etat de l'Art

On ne peut évaluer ses compétences
dans un domaine que l'on ne connaît pas.
[Jean-Louis Etienne]

Chapitre I

Solutions de sécurité pour les systèmes distribués

Plusieurs solutions existent dans l'état de l'art pour résoudre divers problèmes de sécurité pour les systèmes distribués. La plupart de ces solutions sont basées sur la séparation des préoccupations, en séparant le traitement de la sécurité dans des modules ou services distincts. Nous présentons dans cette partie deux types de solutions :

- Les solutions basées sur la configuration de la sécurité à un haut niveau d'abstraction, en particulier au niveau du modèle.
- Les solutions offrant des modules distincts pour l'application des mécanismes de sécurité pendant l'exécution.

1 Configuration de la sécurité à haut niveau d'abstraction

Plusieurs solutions se basant sur la configuration de la sécurité à haut niveau ont été développées. Ces travaux, comme le nôtre, s'appuient sur une configuration séparée du code pour décrire les propriétés non fonctionnelles comme la sécurité et génèrent les codes appropriés. Nous en citons principalement deux : SecureUML et JASON.

1.1 SecureUML

[Basin06] définissent *SecureUML*, une solution orientée modèle qui fait partie de l'approche MDS(*Model Driven Security*). C'est un langage de modélisation de la sécurité qui étend UML

pour représenter les besoins de contrôle d'accès, qui généralisent la notion de contrôle d'accès à base de rôles (RBAC[1] pour *Role-Based Access Control*). À partir d'une description en *SecureUML*, des infrastructures RBAC pour des applications sont automatiquement générées, utilisant les mécanismes de sécurité des plateformes cibles choisies, dont les EJBs (*Enterprise Java Beans*), les *Enterprise Services for .NET* et les *Java Servlets*.

1.2 JASON

[Chmielewski08] définissent JASON, un canevas muni d'un compilateur permettant de simplifier l'application des politiques de sécurité aux services. Le programmeur utilise des annotations Java pour représenter les politiques de sécurité dans son code et les outils du canevas génèrent le code de sécurité adéquat. Le programmeur peut se concentrer simplement sur l'application métier, alors que le canevas JASON s'occupe de générer le code cryptographique nécessaire. JASON supporte plusieurs politiques de sécurité, dont la confidentialité, l'intégrité et RBAC.

2 Modules et services de sécurité

Pour résoudre les problèmes de sécurité à l'exécution, plusieurs solutions incluent l'ajout de modules et services ciblés pour la sécurité du système. Pour la séparation des préoccupations, chaque module de sécurité est défini à l'extérieur des modules fonctionnels.

2.1 Service d'authentification

Ce service, défini par [Welch03] comme étant un service de calcul des créances *Credential Processing Service*, s'occupe des détails du calcul et validation des éléments d'authentification. Il traite du formatage et du calcul des jetons d'authentification pour les échanges avec le service cible. Il soulage l'application et son environnement d'accueil du besoin de comprendre les détails des différents mécanismes.

Dans [Nikander99], ce service est représenté par une couche d'authentification qui peut créer, modifier et supprimer les associations et contextes de sécurité dynamiques entre les autorités. Il est mis en œuvre grâce au protocole de gestion des clefs et de la sécurité internet (ISAKMP pour *Internet Security And Key Management Protocol*). ISAKMP est un protocole permettant d'établir des associations de sécurité et des clefs de cryptographie dans un environnement

1. **RBAC** : Politique de sécurité consistant à définir un ensemble de rôles pour les utilisateurs et à attribuer les permissions aux rôles plutôt que directement aux utilisateurs.

internet. Il fournit un canevas pour l'authentification et l'échange de clefs et est conçu pour être indépendant du protocole d'échange de clefs (les protocoles IKE (*Internet Key Exchange*) ou KINK (*Kerberized Internet Negotiation of Keys*) peuvent être utilisés par exemple).

2.2 Systèmes basés sur le contrôle d'accès

2.2.1 CAS : Community Authorization Service

[Welch03] définit CAS (*Community Authorization Service*) comme étant un service qui évalue les règles de la politique concernant la décision d'autoriser les actions entreprises, en se basant sur des informations sur le demandeur, la cible et la requête. Ce service autorise une politique flexible et expressive à être créée en prenant en considération plusieurs utilisateurs d'une même organisation virtuelle (VO pour *Virtual Organization*). Il permet à une VO d'exprimer la politique qui lui a été délivrée par les fournisseurs de la ressource dans la VO.

Dans le but de se connecter et d'utiliser une ressource, l'utilisateur doit réaliser les actions suivantes :

- Il s'authentifie à CAS et reçoit les assertions exprimant la politique de la VO en termes d'utilisation des ressources.
- Il présente l'assertion à la ressource de la VO avec la requête d'utilisation
- En évaluant si la requête sera autorisée, la ressource vérifie la politique locale et la politique de la VO exprimée dans l'assertion CAS.

2.2.2 PolicyMaker

[Blaze99] présente PolicyMaker, un moteur de gestion de confiance qui adresse le problème d'autorisation directement, plutôt que de le traiter indirectement via l'authentification et le contrôle d'accès. Dans le PolicyMaker, les créances et les politiques sont des assertions représentées comme des paires d'autorité source et de programme décrivant la nature de l'autorité à accorder et des parties à qui elle est accordée.

Les assertions peuvent être :

- Des assertions de politiques : *source = POLICY*. L'ensemble des assertions de politiques fournies au PolicyMaker représente la "racine de confiance" qui définit la décision sur la requête.
- Des assertions de créances : *source = clef publique de l'autorité émettrice*. Ces assertions doivent être signées par leurs émetteurs et ces signatures sont vérifiées avant l'utilisation des créances.

Avec PolicyMaker, c'est l'application qui est responsable de toutes les vérifications cryptographiques des signatures sur les créances et les requêtes.

Le module de gestion de la confiance reçoit comme entrée le triplet (r,C,P), disant que l'ensemble des créances C contient une preuve que la requête r se conforme avec la politique P.

2.2.3 Keynote

[Blaze99] présente KeyNote, un moteur qui suit les mêmes principes que le PolicyMaker et a comme objectif la standardisation et la facilité d'intégration. Il attribue plus de responsabilité au module de gestion de la confiance que le PolicyMaker et moins à l'application appelante. Ainsi, la vérification de la signature cryptographique est réalisée par le moteur de gestion de confiance. Les créances et politiques doivent être écrits dans un langage d'assertion spécifique

L'application appelante passe à un évaluateur KeyNote :

1. Une liste de créances, politiques et clés publiques.
2. Un environnement d'action : une liste de paires attribut/valeur, construits par l'application appelante et contenant toutes les informations considérées appropriées à la requête et nécessaires pour la décision de confiance.

Le résultat de cette évaluation est une chaine de caractères propre à l'application, passé à l'application (comme la chaine "autorisé", par exemple).

Les assertions KeyNote sont structurées de manière à ce que le champ *Licencees* définisse explicitement l'autorité ou les autorités auxquelles un pouvoir est délégué, et que le champ *Conditions* représente le test sur les variables d'environnement d'action.

2.2.4 Service de gestion de la confiance et des politiques

Dans [Nikander99], cette couche permet aux utilisateurs et administrateurs de définir explicitement :

1. Quels nœuds sont de confiance pour quelles opérations ?
2. Comment est-ce que les utilisateurs sont autorisés à accéder aux ressources ?
3. Quel type de protection est requis de la connexion entre agents ?

Les données sont représentées comme des certificats stockés dans le dépôt des certificats. Les certificats contrôlent et facilitent les opérations du protocole d'authentification. Ce module fournit les contextes de sécurité initiaux. Un contexte de sécurité est une collection de variables de sécurité tel que les clefs symétriques ou asymétrique, les règles de politique et les créances,

partagées par deux ou plus de parties. En créant un nouveau contexte de sécurité, l'authenticité, la confidentialité et l'intégrité de l'information doivent être assurées.

Ce module est implémenté via SPKI (*Simple Public Key Infrastructure*). Une autorité a au moins une clé de cryptographie privée en sa possession, et au moins une clé publique qui agit comme étant son identité. Normalement, chaque clé publique représente un rôle appliqué par l'autorité.

2.3 Systèmes à base de composants sécurisés

2.4 SCA

La spécification SCA (présentée dans la section II.2.2, partie I), offre une technique pour attribuer les contraintes de sécurité à haut niveau aux systèmes à base de composants, en sécurisant les liaisons entre les composants fonctionnels. En effet, cette spécification offre la possibilité aux concepteurs de spécifier la politique de sécurité à appliquer à la liaison au niveau du modèle, grâce aux *intents* et aux *policy sets*.

2.4.1 CRACKER

Une extension de Think [2] [Fassino02] pour la constructions de systèmes sécurisés, nommée CRACKER [Lacoste08] a été développée. C'est une architecture de contrôle d'accès à base de composants. L'accès à un composant par un acteur est soumis à l'**autorisation** d'un contrôleur spécifique de ce composant. Cette autorisation est en fonction d'un contexte, des identifiants du composant et des acteurs en question. L'architecture CRACKER définit deux composants principaux :

- ***Reference Monitor* (RM)** : Module implémenté dans la membrane du composant, qui intercepte les appels entrants et appelle le *policy manager* du système pour autoriser ou refuser l'accès au composant contrôlé.
- ***Policy Manager* (PM)** : C'est le gestionnaire de la politique d'accès. Il implémente la matrice de contrôle d'accès (*acteurs/composants*) et fournit une interface utilisée par les RM lors de l'autorisation d'un accès, et des interfaces pour l'administration du système.

2. **Think** : Projet lancé en 1998 par France Télécom R&D pour développer un canevas logiciel pour la construction de noyaux de systèmes d'exploitation à base de composants. C'est l'implémentation en C du modèle Fractal.

2.4.2 CAmkES

CAmkES [Kuz07], une architecture à base de composants pour le développement de systèmes embarqués à micro-noyau, offre un mécanisme de contrôle d'accès en se basant sur le modèle de **capacité** fourni par le système d'exploitation L4/Iguana [Heiser05].

CAmkES utilise le concept de *Configuration Specification* pour représenter la liste des capacités correspondant à une connexion (ou *liaison* entre deux composants). Ces capacités représentent les autorisations données à chaque interface du composant pour accéder aux méthodes d'autres interfaces (correspondant aux *ports* du composant).

Une fois la stratégie de sécurité définie, son application est faite à l'exécution par Iguana, qui fournit les outils nécessaires pour cela.

2.5 Modules cryptographiques

2.5.1 Gestion de clefs

DKMS (*Distributed Key Management System*) [3] offre un service de gestion des certificats et des clefs symétriques et asymétriques. Il permet de gérer plusieurs serveurs à partir d'une console unique : la station de travail DKMS. Elle est connectée aux serveurs équipés d'engins cryptographiques et hébergeant les certificats ou clefs. L'un des serveurs détient un dépôt de clefs DKMS centralisé utilisé pour sauvegarder toutes les clefs et tous les certificats.

Voltage [4] emploie une architecture de gestion de clefs à base d'identité. Les clefs sont munies de noms, représentant une *identité* qui peut être utilisée pour référencer facilement la clef appropriée. Les noms sont de la forme *identifiant@domaine* et peuvent représenter un utilisateur, un groupe ou même une politique complexe. Ce modèle de nommage est utilisé pour le chiffrement, où le nom peut être mathématiquement converti en clef publique, de même que pour le chiffrement symétrique où le nom est utilisé pour dériver la clef symétrique. Voltage utilise un modèle d'authentification qui permet aux utilisateurs, systèmes et applications d'être authentifiés en utilisant presque n'importe quel mécanisme.

[Seitz03] décrit un ensemble de schémas pour le stockage de clefs cryptographiques en utilisant la gestion de clefs sécurisée. Dans ces travaux, un serveur de clefs doit être utilisé dans le cas où les permissions sur les ressources sont évaluées à l'exécution. Pour améliorer la disponibilité des clefs de cryptographie, les auteurs proposent de distribuer les clefs sur plusieurs serveurs. Pour réduire la vulnérabilité, ils proposent de distribuer une même clef sur plusieurs serveurs de clefs.

3. **IBM Distributed Key Management System** : http ://www-03.ibm.com/security/products/prod_dkms.shtml

4. **Voltage** : http ://www.voltage.com/technology/key-management.htm

2.5.2 Conversion de créances

[Welch03] définit un service qui permet de faire le relai entre différents domaines de confiance en convertissant les créances entre les racines ou mécanismes de confiance. Il convertit les créances existantes au format ou mécanisme approprié.

2.5.3 Mise en correspondance des identités

[Welch03] définit un service qui prend en entrée une identité d'un utilisateur dans un domaine et qui retourne son identité dans un autre. Par exemple, il donne le nom Kerberos associé à l'identité X509.

3 Étude comparative entre les solutions pour la sécurité des systèmes distribués

Le tableau I.1 permet de comparer les différentes solutions que nous avons présentées selon les critères qui, à notre avis, sont importants pour représenter la politique de sécurité dans les systèmes distribués :

Configuration des propriétés de sécurité à haut niveau Ce critère permet à l'administrateur de représenter ses contraintes et de spécifier sa politique sans entrer dans les détails d'implémentation du système. Nous remarquons ici que la plupart des solutions permettent la configuration de la sécurité à haut niveau, en utilisant des listes ou des annotations pour affecter les contraintes de sécurité. Seul JASON ne le permet pas, puisque l'administrateur doit annoter tout le code, même les variables intermédiaires, ce qui peut s'avérer ardu.

Génération de code cryptographique Ce critère est important car il est parfois difficile pour le programmeur d'écrire à la main le code nécessaire pour réaliser les opérations de cryptographie, surtout que ce code est en général redondant. Il serait donc intéressant de le générer automatiquement selon certaines configurations exprimées au préalable. D'après le tableau comparatif, peu de solutions proposent la gestion automatique du code de cryptographie.

Séparation des préoccupations Ce critère consiste en la possibilité de séparer la configuration de sécurité du code fonctionnel. Cette séparation est utile si le concepteur du système et l'administrateur sont deux personnes différentes. Ainsi, chacun d'entre eux gère la partie qui le concerne, et la solution s'occupe de les faire coïncider. La plupart des solutions proposent de

séparer l'implémentation de la sécurité dans des modules externes, sauf les solutions de sécurité à base de modèle (JASON et SecureUML) qui intègrent la configuration de sécurité directement dans l'implémentation.

Reconfiguration dynamique La reconfiguration dynamique est un critère très important à considérer, puisque les systèmes que nous ciblons, soit les systèmes distribués, sont hautement dynamiques. La solution de sécurité ciblée doit donc garantir que la modification de l'architecture du système pendant l'exécution ne va pas nuire aux propriétés de sécurité définies à la compilation. La reconfiguration dynamique n'est pas prise en compte par toutes les solutions.

Tableau I.1 – Tableau comparatif des solutions de sécurité pour les systèmes distribués

	Configuration haut niveau	**Génération de code crypto.**	**Séparation des préoccupations**	**Reconfiguration dynamique**
JASON		X		
SecureUML	X			
SCA	X	X	X	X
CRACKER	X		X	X
CAmkES	X		X	X
CAS	X		X	X
PolicyMaker	X		X	
Keynote	X	X	X	

Nous remarquons dans le tableau précédent que la seule solution qui permette de satisfaire les quatre critères présentés est la spécification SCA. En effet, cette solution permet une configuration de la sécurité à un haut niveau en allouant des intentions aux interfaces des composants ; ces intentions sont ensuite traduites par l'implémentation sous-jacente en code cryptographique assurant ces intentions définies à haut niveau. La politique de sécurité est spécifiée dans un fichier de configuration séparé appelé *definitions.xml*, ce qui permet une séparation des préoccupations. Enfin, SCA permet la reconfiguration dynamique, car toute connexion avec un nouveau composant doit satisfaire les intentions définies aux deux bouts de la liaison. Cependant, SCA ne permet pas d'assurer le contrôle de flux d'information, comme le reste des solutions proposées d'ailleurs.

Conclusion

Les solutions de sécurité appliquées aux systèmes distribués que nous avons présentées dans ce chapitre permettent de satisfaire plusieurs critères nécessaires pour garantir la construction et

exécution d'applications sécurisées. Cependant, en plus des propriétés d'authentification, confidentialité et intégrité usuellement abordées, il faudrait appliquer un contrôle de flux rigoureux. Nous présentons donc dans le chapitre suivant les solutions présentes dans la littérature pour appliquer le contrôle de flux d'informations et en particulier la propriété de non-interférence.

Chapitre II

Systèmes de contrôle de flux d'information

Dans ce chapitre, nous répartissons les solutions de contrôle de flux en deux types : les solutions garantissant un contrôle de flux d'information **statique**, c'est à dire uniquement à la compilation, et les solutions garantissant un contrôle de flux **dynamique**, ou pendant l'exécution. Nous énumérons plusieurs solutions existantes dans la littérature, en les comparant selon plusieurs critères, dont la granularité ou le support de modules patrimoniaux.

1 Solutions de vérification statique de la non-interférence

L'analyse de flux d'information consiste en une analyse statique du code source d'un programme *avant* son exécution, de manière à assurer que toutes les opérations qu'il réalise respectent la politique de sécurité du système [Simonet03]. Cela implique le suivi de chaque flux d'information du système et la vérification de sa légitimité.

Cette analyse de flux a surtout été appliquée sur les systèmes centralisés, mais certains travaux l'ont adapté aux systèmes distribués.

1.1 Solutions centralisées

1.1.1 JIF : Java Information Flow

JIF [1] [Myers00] est une extension du langage de programmation Java basée sur le langage JFlow [Myers99]. Il ajoute une analyse statique du flux d'information pour une vérification améliorée de la sécurité. Le but principal de JIF est d'empêcher les informations d'être utilisées de manière impropre.

1. **JIF** : www.cs.cornell.edu/jif/

JIF étend Java en ajoutant des étiquettes (suivant le modèle d'étiquettes décentralisé présenté dans la section III.2.1, partie I) qui expriment des restrictions sur la manière dont l'information peut être utilisée. En utilisant ces annotations, le compilateur JIF (basé sur le compilateur Polyglot[Nystrom03]) analyse le flux de l'information dans les programmes et détermine si les politiques de sécurité pour la confidentialité ou l'intégrité de l'information sont renforcées par le programme.

Si un programme JIF est correctement typé, le compilateur le traduit en code Java qui peut être compilé avec un compilateur Java standard, puis exécuté avec une machine virtuelle Java standard.

JIF définit plusieurs notions, que nous décrivons brièvement dans ce qui suit.

Types étiquetés Dans JIF, chaque valeur a un type étiqueté qui consiste en deux parties : un type Java ordinaire (*int* par exemple), et une étiquette qui décrit la manière dont la valeur peut être utilisée. Par exemple :

$$int\{Alice \rightarrow\}\ x = 2;$$

représente la déclaration et l'initialisation de la variable x avec le type étiqueté $int\{Alice \rightarrow\}$. La variable x est donc un entier public, dont le propriétaire est l'autorité *Alice*.

Compteur programme Chaque point dans le programme admet une étiquette appelée compteur programme (ou *pc* pour *program counter*). Pour chaque point du programme donné, *pc* est une limite supérieure sur les informations qui peuvent être déduites par le fait de savoir que l'exécution a atteint ce point. De manière équivalente, *pc* peut être considéré comme une limite supérieure des étiquettes de toutes les valeurs qui ont fait que le flux de contrôle du programme permette d'atteindre ce point du programme.

Méthodes La déclaration des méthodes dans JIF étend la syntaxe Java en ajoutant plusieurs annotations pour le contrôle de flux d'information et la gestion des autorités. Dans la déclaration d'une méthode JIF, la valeur de retour, les arguments et les exceptions peuvent être annotés avec une étiquette. Il existe également deux étiquettes optionnelles dans la déclaration d'une méthode :

- **L'étiquette de début** (*begin label*) : spécifie une limite supérieure à la valeur du *pc* à l'invocation de la méthode. Il permet à l'information sur le *pc* de l'appelant d'être utilisée pour vérifier statiquement l'implémentation et empêcher ainsi la création de flux implicites à l'intérieur de la méthode.

– **L'étiquette de fin** (*end label*) : spécifie la valeur du *pc* au point de terminaison de la méthode, et représente une limite supérieure sur l'information qui peut être apprise en observant si la méthode se termine normalement ou si elle génère des exceptions.

Étiquettes et autorités dynamiques Même si le contrôle du flux d'information est fait la plupart du temps à la compilation, JIF permet également d'en renforcer une partie à l'exécution grâce aux étiquettes et autorités dynamiques. Une étiquette ou autorité dynamique peut être utilisée comme paramètre dans une classe, dont la valeur est déterminée à l'exécution, au moment de l'instanciation d'un objet de cette classe.

Exemple L'extrait de code II.1 représente la version JIF de la classe *Vector* de Java [2].

Listing II.1 – Extrait de la classe *Vector.jif*

```
public class Vector[label L] extends AbstractList[L] {
  private int{L} length;
  private Object{L}[]{L} elements;
  public Vector() ...
  public Object elementAt(int i):{L;i}
        throws IndexOutOfBoundsException {
    ...
    return elements[i];
  }
  public void setElementAt{L}(Object{L} o, int{L} i) ...
  public int{L} size() {
    return length;
  }
  public void clear{L}() ...
  ...
}
```

La classe *Vector* est une classe générique en ce qui concerne le paramètre (étiquette) **L**, ce qui permet aux vecteurs d'être utilisés pour stocker des informations sur n'importe quelle étiquette. La méthode *setElementAt* a une étiquette de début $\{L\}$, placée entre le nom de la méthode et la liste d'arguments. Cette étiquette assure que la méthode peut être appelée à partir d'une autre méthode seulement si le *pc* de la méthode appelante n'est pas plus restrictif que $\{L\}$. Il assure également que la méthode peut uniquement modifier des données dont l'étiquette est au

2. Extrait du **manuel de référence de JIF**, http ://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html, version du 01/02/2009

moins aussi restrictive que $\{L\}$ (tel que la variable *elements*). Ces deux restrictions combinées assurent qu'il n'y aura pas de flux d'information implicite via l'invocation de la méthode. Les paramètres formels o et i de la méthode *setElementAt* portent l'étiquette $\{L\}$, ce qui veut dire que $\{L\}$ est une limite supérieure sur les étiquettes des paramètres effectifs. L'étiquette de fin de la méthode *elementAt*, $\{L; i\}$ veut dire aue le *pc* résultant de la terminaison normale de la méthode est au moins aussi restrictif que le paramètre L et que l'étiquette de l'argument i à la fois. Cette étiquette de fin est nécessaire car, quand l'exception *IndexOutOfBoundsException* est lancée, la variable *elements* et l'argument i sont divulgués. Ainsi, la connaissance du chemin de terminaison de la méthode peut donner des informations sur le contenu de ces deux variables.

Plusieurs travaux ont été réalisés sur le langage JIF. [Zdancewic03] définit un système de type pour représenter le langage JIF. [Myers06, Chong06] étudient la propriété de robustesse de JIF, qui permet d'assurer que la décision de rétrogradation de l'information n'est pas influencée par un attaquant. [Chong07] définit un canevas qui utilise JIF pour construire des applications web avec contrôle de flux d'information et [Zdancewic02] décrit une technique pour construire des systèmes distribués non-interférents.

Nous nous sommes inspirés du système de type de JIF pour déterminer le comportement du compilateur pour la vérification intra-composant de la non-interférence. Nous expliquons dans la section I.1.2, partie III comment est-ce que le compilateur Polyglot est utilisé pour propager les étiquettes dans le code d'un composant.

1.1.2 FlowCaml

Flow Caml [Simonet03] est une extension du langage Objective Caml pour le suivi du flux d'information dans le code. Il permet d'écrire des programmes et de vérifier automatiquement qu'ils respectent certaines politiques de sécurité. Les types simples sont annotés grâce à des étiquettes de sécurité et le système vérifie, grâce à une inférence de type, que les flux d'information dans le programme respectent la politique de sécurité spécifiée, sans avoir besoin d'annoter tout le code source.

1.2 Solutions distribuées

Les solutions présentées auparavant concernent principalement les systèmes centralisés, et ne traitent pas les problèmes que peut poser la distribution du programme sur un réseau non fiable. Certaines de ces solutions, comme JIF, ont été étendues pour s'adapter aux systèmes distribués.

1.2.1 JIF/Split

[Zdancewic02] décrit JIF/Split, une implémentation du concept de **partitionnement sécurisé des programmes**, une technique à base de langage permettant de protéger les données confidentielles durant le calcul dans les systèmes distribués contenant des hôtes mutuellement hostiles. Les politiques de confidentialité et d'intégrité peuvent être exprimées en annotant les programmes avec des types de sécurité qui contraignent le flux d'information, en utilisant le modèle DLM.

L'outil JIF/Split permet de partitionner automatiquement ce programme −initialement centralisé− pour créer automatiquement un ensemble de sous-programmes non-interférents. Ces programmes peuvent ainsi être exécutés de manière sécurisée sur des hôtes hétérogènes. Les sous-programmes résultants mettent en œuvre collectivement le programme originel, avec en plus la satisfaction des exigences de sécurité des autorités participantes sans avoir besoin d'une machine universellement sûre.

1.2.2 Compilateur de [Fournet09]

[Fournet09] ont récemment présenté la même approche que JIF/Split, mais le compilateur renforce en plus la sécurité des communications en ajoutant des mécanismes cryptographiques. Ce compilateur est construit pour un langage impératif simple avec des annotations de sécurité et du code distribué lié à des bibliothèques cryptographiques concrètes.

Le compilateur présenté ici est structuré en quatre étapes. La première est le *partitionnement*, elle permet de découper le code séquentiel en un ensemble de sous-programmes en suivant les annotations appliquées par le concepteur de sécurité. La deuxième étape est le *contrôle de flux* qui permet de protéger le programme contre un ordonnanceur malicieux, en générant du code qui garde la trace de l'état de ce programme. L'étape de *réplication* transforme un programme distribué basé sur une mémoire partagée en un programme où les variables sont répliquées à chaque nœud du système. Finalement, la dernière étape est la *cryptographie*, qui insère des opérations de cryptage pour protéger ces réplications et génère un protocole initial pour distribuer leurs clefs.

1.2.3 Fabric

Fabric [Liu09] est une plateforme pour la construction de systèmes distribués sécurisés qui favorise le partage sécurisé de ressources entre nœuds hétérogènes qui ne se font pas nécessairement confiance. Les nœuds sont répartis en nœuds de stockage (*storage nodes*), de dissémination (*dissemination nodes*) et de calcul (*worker nodes*). Ces nœuds partagent un ensemble d'objets

Fabric, qui sont similaires à des objets Java et qui ont chacun une étiquette de sécurité immuable, suivant le modèle d'étiquettes décentralisé DLM.

La plateforme fournit un outil de déploiement renforcé avec la réplication pour la tolérance aux pannes. Le langage Fabric est une extension de JIF pour la programmation distribuée, avec un support pour les transactions et pour l'appel de méthode distant.

1.2.4 Solution de [Alpizar09]

[Alpizar09] présente un langage impératif abstrait simple pour les systèmes distribués, qui permet de dissimuler les protocoles de communication et les opérations cryptographiques entre les processus. Ce langage définit des mécanismes pour :

- Envoyer et recevoir des données,
- Classifier les données selon leur niveau de sécurité,
- Définir la propriété de non-interférence.

Le système distribué est représenté par un ensemble de **processus** exécutant des programmes sur des nœuds différents. Chaque processus a ses propres **mémoires** et est capable d'envoyer (primitive *send*) et de recevoir (primitive *receive*) des messages provenant d'autres processus grâce à des **canaux de communication** distincts.

Les données sont classifiées selon un treillis de sécurité simplifié, limité aux niveaux de sécurité H (pour *High*, représentant les données privées) et L (pour *Low*, représentant les données publiques). À chaque donnée est associée une étiquette de sécurité, représentant son niveau de confidentialité (le niveau d'intégrité n'étant pas pris en considération dans ce travail).

Les canaux de communication utilisés pour la transmission de données doivent satisfaire les contraintes suivantes :

- Chaque canal doit avoir une classification de sécurité. Pour pouvoir transmettre des données avec différents niveaux de confidentialité, des canaux séparés doivent être définis pour chaque classification.
- Chaque canal doit avoir un processus source et un processus cible bien spécifiés. Ainsi, chaque canal peut être affecté d'un type de la forme : $\tau ch_{i,j}$ avec τ représentant le niveau de sécurité, i le processus source et j le processus cible.
- Les processus ne doivent pas être capables de tester si un canal contient des données, car cela pourrait autoriser une fuite d'information grâce aux *canaux temporels*. En effet, la connaissance du temps nécessaire pour qu'une étape d'exécution ait lieu peut être une donnée précieuse responsable d'une fuite d'information.

– À chaque *receive* doit correspondre un *send* sur le même canal, car la réception est bloquante. Le contraire n'est, par contre, pas obligatoire ; chaque canal a donc besoin d'un **tampon** de taille illimitée pour stocker les messages sortants en attente d'être reçus.

Pour appliquer la non-interférence sur ce langage, les auteurs utilisent uniquement les restrictions de Denning [Denning77] qui contrôlent le flux d'information illégal dans deux types d'instructions uniquement :

– Une affectation $l := e$ n'est pas autorisée si l est une variable publique alors que l'expression e contient des variables privées.
– Une affectation à une variable publique n'est pas autorisée si elle est faite dans un bloc **if** ou **while** dont la garde (expression conditionnelle) contient des variables privées.

De plus, pour sécuriser l'échange de données dans un réseau qui n'est pas de confiance, la cryptographie asymétrique est utilisée.

Ce travail prouve que tout système distribué respectant les contraintes précédentes est non-interférent, car deux ensembles d'entrées L-équivalentes (c.-à-d. dont les données de niveau L sont équivalentes) produisent des sorties indistinguables au niveau L.

Les solutions proposées ci-dessus offrent une vérification de la non-interférence en utilisant l'analyse de flux, mais ne permettent pas (ou peu) de réaliser cette vérification pendant l'exécution. En effet, les langages et outils proposés sont uniquement déployés à la compilation. Nous proposons dans la partie suivante des solutions existantes pour la vérification dynamique de la non-interférence.

2 Solutions de vérification dynamique de la non-interférence

Le contrôle de flux d'information statique permet de créer des systèmes non-interférents par construction. Cependant, les systèmes distribués sont sujets à une modification continue pendant l'exécution. En effet, dans certains cas, des données voient leurs niveaux de sécurité changer selon le contexte de leur utilisation. Dans d'autres cas, c'est l'architecture du système qui évolue, certains composants pouvant migrer ou tomber en panne. Dans ces cas, l'application doit adapter son comportement à ces changements : la configuration de sécurité doit être revérifiée.

Nous présentons ci-dessous des solutions présentes dans l'état de l'art, proposant des langages ou plateformes pour la vérification dynamique de la non-interférence.

2.1 Contrôle de flux d'information dans les systèmes d'exploitation

Certains systèmes d'exploitation ont été définis pour assurer le contrôle du flux d'information entre les processus, tel que Flume [Krohn07], HiStar [Zeldovich06] et Asbestos [Efstathopoulos05], et ce en associant des étiquettes de sécurité aux processus et messages. Ces systèmes utilisent le contrôle de flux d'information distribué DIFC (*Distributed Information Flow Control*) pour contrôler la manière dont les données circulent entre les entités d'une application et le monde extérieur. Ils permettent aux utilisateurs de sécuriser les applications existantes en associant des étiquettes aux processus et messages et en régulant la communication et le changement d'étiquettes de ces processus. Ils visent surtout à minimiser la quantité du code auquel on doit faire confiance, et permettent aux utilisateurs de spécifier des politiques de sécurité précises sans limiter la structure des applications.

2.2 Solutions distribuées

2.2.1 DStar

[Zeldovich08] présente DStar, un système qui fournit une protection au niveau du système d'exploitation sur des machines mutuellement hostiles en utilisant le contrôle de flux d'information distribué DIFC. Pour spécifier les permissions, il associe les étiquettes aux processus et aux messages. Pour communiquer entre les machines, DStar introduit un *exporteur* par machine, le seul processus qui puisse communiquer avec d'autres processus à travers le réseau et qui vérifie que l'échange d'information entre les processus distants n'enfreint pas les restrictions de sécurité définies dans les étiquettes.

DStar a été développé pour tourner sur plusieurs systèmes d'exploitation, dont HiStar, Flume et Linux. L'inconvénient principal de DStar est que, s'il s'exécute sur un système d'exploitation ne supportant pas la DIFC tel que Linux, il doit faire confiance à tous les logiciels qui tournent dessus.

2.2.2 SmartFlow

Le projet SmartFlow [Eyers09] cherche à développer un intergiciel extensible et distribué utilisant la DIFC pour sécuriser les systèmes à base d'évènements. Les systèmes à base d'évènements sont des systèmes qui transmettent des **évènements** entre des composants logiciels et services faiblement couplés, en partant d'un émetteur d'évènements (*agent*) vers un consommateur d'évènements (*sink*). Un évènement peut être défini comme un changement significatif d'état pour un objet.

SmartFlow définit un intergiciel basé sur les évènements qui intègre de manière sécurisée les systèmes hétérogènes et fournit un canevas pour gérer automatiquement les extensions middleware. SmartFlow permet d'affecter des étiquettes aux données et aux processus et de contrôler ainsi leur consultation ou altération.

2.2.3 Composition de services web

La composition des services web est le processus permettant de créer un service composite à partir de services web existants plus simples. [Hutter06] propose une méthodologie pour contrôler la sécurité des données dynamiquement calculées dans les services web composés, en utilisant les techniques de contrôle de flux d'information. Chaque donnée est équipée d'un type spécifiant sa classification par rapport à plusieurs catégories de sécurité : cette classification détermine l'accessibilité de la donnée par les services web individuels.

Pour prouver que le système ne viole pas la politique de sécurité définie, ce travail expose un langage de programmation typé simplifié, étendu avec des appels aux services web et utilisant le calcul de type (*type calculus*) pour propager les types de données.

Chaque donnée est équipée d'un type qui spécifie sa classification selon des catégories de sécurité définies par l'utilisateur. Ainsi, les services web peuvent seulement être utilisés dans la composition si et seulement s'ils ont l'habilitation nécessaire pour manipuler l'information qu'ils reçoivent en exécutant le service. De plus, le calcul de toute donnée publique doit toujours être indépendant des données privées. Quand une nouvelle donnée est produite, elle est automatiquement étiquetée selon les classifications des données utilisées pour la calculer.

3 Étude comparative entre les solutions de contrôle de flux d'information

Le tableau II.1 permet de comparer les solutions précédentes selon les critères nécessaires pour un contrôle du flux d'information efficace et facile à appliquer. Ces critères sont, essentiellement :

Dynamicité Le contrôle du flux d'information est-il assuré pendant la reconfiguration dynamique du système ? La dynamicité est un critère très important dans les systèmes distribués dont l'architecture subit plusieurs changements pendant l'exécution. Ainsi, Flume, DStar et SmartFlow sont des solutions dynamiques, alors que les autres assurent le contrôle de flux d'information à la compilation uniquement, avec une petite touche de dynamicité pour les solutions basées sur JIF, qui ajoutent les notions d'étiquettes et autorités dynamiques.

Granularité Le contrôle de flux d'information peut se faire à la granularité de la variable, de l'objet ou du processus. Plus la granularité est petite, plus le contrôle est précis, mais son application ardue. Ainsi, seules les solutions de contrôle de flux d'information pour les systèmes d'exploitation tel que Flume réalisent le contrôle à gros grains, car ils affectent les étiquettes de sécurité aux processus et messages. Par contre, toutes les solutions basées sur l'analyse de flux d'information dans le code assurent un contrôle à grain très fin. Il est cependant à noter que FlowCaml est la seule solution qui permette d'assurer un contrôle à grain très fin en utilisant une annotation à haut niveau, puis une inférence de type pour propager automatiquement les étiquettes dans le code. Les autres solutions utilisant un contrôle de flux à grain très fin exigent que le développeur affecte les étiquettes manuellement à toutes les entités (variables, objets...) utilisées dans le code.

Support de modules patrimoniaux Le support des modules patrimoniaux indique le niveau de flexibilité de la solution. Ainsi, les solutions qui n'autorisent pas la prise en compte automatique des modules réalisés par autrui et importés dans le système posent problème quant à leur facilité d'utilisation et leur adaptabilité aux systèmes existants. Ainsi, seul SmartFlow est applicable à des systèmes patrimoniaux hétérogènes. Les autres solutions obligent le développeur à utiliser un langage ou une plateforme particulière.

Support de la cryptographie Ce critère indique si la solution propose un support automatique pour les mécanismes de cryptographie nécessaires pour le transport des données à travers les réseaux non sécurisés. Une solution qui ne gère pas la couche de cryptographie se décharge de son application sur le développeur du système.

Systèmes cibles Ce critère nous indique si la solution proposée prend en compte les contraintes posées par les systèmes distribués, ou si elle est applicable uniquement pour les systèmes centralisés. De plus, elle précise si la solution a pour cible des systèmes particulier, comme par exemple SmartFlow qui sécurise uniquement les systèmes à base d'évènements.

D'après les résultats relevés, aucune de ces solutions ne peut satisfaire tous les critères à la fois. Nous remarquons surtout que dynamicité est souvent négligée au dépend de la granularité. En effet, il est assez délicat de gérer un flux d'information à grain très fin pour les systèmes distribués, si l'aspect dynamique du système doit être pris en considération. D'autre part, très peu de solutions supportent les systèmes patrimoniaux existants, car le contrôle de flux, surtout à grain fin, est très proche du code ; il faut donc réfléchir à une solution qui permette d'appliquer ce contrôle a posteriori si le code du module est disponible (ce qui n'est pas toujours le cas).

Tableau II.1 – Tableau comparatif des solutions de CFI

	Dynamicité	**Granularité**	**Support sys. patrimoniaux**	**Support crypto.**	**Systèmes cibles**
JIF	statique (+ étiq. dyn.)	très fine	non	non	centralisé
FlowCaml	statique	très fine (inférence de type)	non	non	centralisé
JIF/Split	statique (+ étiq. dyn.)	très fine	non	non	distribué
[**Fournet09**]	statique	très fine	non	oui	distribué multi-thread
Fabric	statique (+ étiq. dyn.)	fine	non	oui	distribué
Flume	dynamique	grosse	non	non	centralisé
DStar	dynamique	fine	non	oui	distribué
SmartFlow	dynamique	fine	oui	non	distribués à base d'évènements

Conclusion

L'objectif principal de notre travail est d'assurer la non-interférence à granularité fine pour des systèmes distribués dynamiques, avec support pour les composants patrimoniaux et en facilitant le plus possible les techniques d'affectation de la politique de sécurité. Dans cette optique, aucune des solutions que nous avons présenté ci-dessus n'est suffisante. C'est pour cette raison que nous proposons de réaliser un modèle à base de composants générique, muni des outils d'automatisation nécessaires pour le contrôle de flux d'information à la compilation et à l'exécution.

Troisième partie

Contribution : Modèles et Outils pour l'Application Statique et Dynamique de la Non-Interférence

Tout le génie que je peux avoir est simplement
le fruit de la réflexion et du travail.

[Alexander Hamilton]

Chapitre I

CIF : Outils de vérification statique de la non-interférence

Nous présentons dans ce chapitre CIF, un ensemble d'outils que nous proposons pour assurer la non-interférence à la compilation. Ces outils permettent la vérification du flux d'information de manière automatique, et ce à deux niveau : au niveau du composant lui-même et entre les différents composants du système.

1 Sécuriser un système à base de composants avec CIF

CIF (*Component Information Flow*) est un ensemble d'outils pour la configuration, la vérification et la génération de code pour les systèmes à base de composants. Ces systèmes sont représentés selon un modèle répondant aux contraintes décrites dans la section II.1, partie I. Nous présentons dans ce qui suit la configuration de la sécurité sur les systèmes cibles, puis détaillons les outils nécessaires pour automatiser le processus de vérification de la non-interférence à la compilation.

1.1 Configuration de sécurité

Pour le contrôle de flux d'information, les architectures à base de composants facilitent le suivi du flux d'information grâce aux liaisons explicites entre les composants. Dans CIF, les étiquettes de sécurité sont attribuées aux **attributs**, aux **ports serveurs** (entrées) ainsi qu'aux **ports clients** (sorties) de chaque composant. L'attribution des niveaux de sécurité aux différents éléments d'un composant est réalisée dans un fichier de configuration séparé du code et de l'ADL, appelé **Policy**. En effet, la politique de sécurité doit être définie indépendamment de

Fichier ADL SCA

```
<composite name="C" xmlns="...">
  <component name="C1">
    <reference  name="P"   target="C2/P' ">
          <interface.java interface="security.PItf"/>
    </reference>
    <property name="M"> My Message </property>
    <implementation.java class="security.C1Impl" />
  </component>
  <component name="C2">
    <service  name="P' ">
          <interface.java interface ="security.PItf"/>
    </service>
  </component>
</composite>
```

Fichier Policy

```
<policy targetComposite="C">
  <component name="C1">
    <port name="P"   label="{S;I}" />
    <attribute name="M"  label="{Sm;Im}"/>
    <capabilities>
          <capability> cap1 </capability>
    </capabilities>
  </component>
  <component name="C2">
    <port  name="P' " label="{S';I'}" />
  </component>
  <library name="java.lang.Math">
     <method name="sqrt>
          <param index="1" varLabel="x"/>
          <return varLabel="x"/>
     </method>
  </library>
</policy>
```

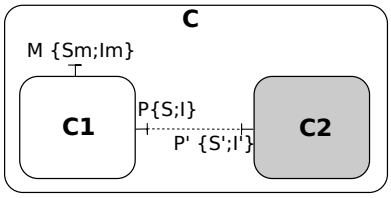


Figure III.I.1 – Configuration des paramètres de sécurité dans le fichier Policy

l'architecture et du code fonctionnel du système car les rôles de développeur et d'administrateur du système ne sont généralement pas attribués à la même personne.

La figure III.I.1 représente un système distribué très simple formé de deux composants : un composant client C_1 et un composant serveur C_2. C_1 envoie un message à C_2 via le port client P, et C_2 le reçoit sur son port P'. C_1 a un attribut M dont la valeur est attribuée dans le fichier ADL, écrit pour notre exemple dans le langage ADL SCA. Le composant C_2 est représenté par un rectangle grisé, car c'est un composant patrimonial (*legacy*), ce qui veut dire dans notre cas qu'il est vu par le développeur du système comme une boîte noire, complètement opaque, sauf pour ses ports externes et ses attributs. Néanmoins, ces composants patrimoniaux sont de confiance ; nous considèrons que leur code implante les spécifications prévues.

Composition du fichier Policy

Dans le fichier de configuration *Policy*, le concepteur attribue des étiquettes de sécurité aux différents ports et attributs des composants du système. Il peut également étiqueter une bibliothèque importée, utilisée dans le code du composant. Attribuer une étiquette à une bibliothèque consiste à détailler les étiquettes des méthodes de cette bibliothèque utilisées dans le code.

Dans la figure III.I.1, nous donnons un exemple de bibliothèque importée : *java.lang.Math*. Nous utilisons sa méthode *sqrt* dans le code du composant C_1. Nous partons de l'hypothèse

que le développeur connait le comportement de la méthode *sqrt* et qu'il a confiance dans le fait qu'elle ne divulgue pas ses informations, ou bien juge que la fuite d'information possible n'aura pas d'impact sur la sécurité de son code. Dans ce cas, il peut ajouter une entrée dans le fichier *Policy* indiquant, comme dans l'exemple, que si la méthode a comme paramètre une variable d'étiquette x, alors elle retournera une valeur portant la même étiquette. Ainsi, le développeur n'aura pas à annoter toutes les méthodes qu'il utilise, mais à indiquer à haut niveau les relations entre les entrées et les sorties de ces méthodes. Dans le cas où il dispose du code de la bibliothèque utilisée, il pourra, bien entendu, vérifier son comportement en utilisant l'outil CIFIntra (voir la section I.2.2, partie III). Cette propriété est très importante en pratique, car elle permet d'importer dans le code d'un composant une bibliothèque de confiance dont nous ne disposons que du binaire.

De plus, le concepteur peut attribuer des **capacités** à un composant, appelées *capabilities* dans le ficher *Policy*. Ces capacités sont utilisées pour associer à chaque composant les privilèges de rétrogradation auxquels il a droit, et l'autoriser ainsi à relâcher les contraintes sur le système en permettant automatiquement certaines interférences. La rétrogradation est expliquée plus en détails dans la section I.2.2.3, partie III.

Signification de l'annotation des ports

Concernant les ports, annoter le port client P du composant C_1 avec l'étiquette $\{S; I\}$ et le port serveur P' du composant C_2 avec $\{S'; I'\}$ implique que :

- Du point de vue de la confidentialité, avoir le niveau S pour le port P veut dire que C_1 voudrait que le message qu'il envoie à travers P garde le niveau de confidentialité S. Attribuer le niveau de confidentialité S' au port P' veut dire que le composant C_2 garantit que le niveau de confidentialité S' est préservé dans le programme.
- Du point de vue de l'intégrité, avoir le niveau I pour le port P veut dire que le composant C_1 garantit que le message qu'il envoie à travers P a le niveau d'intégrité I. Attribuer le niveau d'intégrité I' au port P' veut dire que C_2 exige que le message qu'il reçoit sur P' ait au moins une intégrité égale à I'.

Pour respecter la propriété de non-interférence, nous devons suivre le flux d'information. Grâce au modèle orienté composants que nous utilisons, nous distinguons deux types de flux différents : un flux d'information entre les composants circulant dans un réseau qui n'est pas toujours fiable, et un flux d'information dans le code du composant. Nous traitons chacun de ces flux à part.

1.2 Sécurité intra-composant

La sécurité intra-composant est vérifiée pour chaque composant dont le code est disponible, en appliquant le générateur de code intra-composant sur son implémentation. Cet outil aura pour tâche de vérifier que le code d'un composant ne comporte pas d'interférences illégales selon les contraintes spécifiées par le concepteur de sécurité dans le fichier *Policy*. Il permet de propager les étiquettes dans le code en affectant les étiquettes adéquates aux variables intermédiaires et en vérifiant en même temps le flux d'information circulant entre les différentes variables. Le comportement du générateur est décrit plus en détails dans la section I.2.2, partie III.

1.3 Sécurité inter-composant

La liaison entre un port source P et un port destination P' n'est autorisée que si $label(P) \subseteq label(P')$. Ainsi, la contrainte principale de la non-interférence citée dans la section III.2, partie I est respectée, à savoir que les données sont uniquement envoyées vers des cibles plus restrictives. Pour assurer la sécurité du transport des messages, les étiquettes doivent être implantées dans le système cible en introduisant des primitives cryptographiques. CIF insère les composants de sécurité qui interceptent les données sortantes (respectivement entrantes) d'un composant fonctionnel pour chiffrer et/ou signer (respectivement déchiffrer et/ou vérifier) les messages. Dans notre implémentation, nous utilisons un chiffrement asymétrique fortement sécurisé et une signature digitale pour mettre en œuvre respectivement les besoins de confidentialité et d'intégrité.

2 Les outils CIF

CIF peut s'appliquer sur n'importe quel système modélisé avec un langage orienté composants qui répond aux exigences suivantes :

- *La séparation de l'architecture et de l'implémentation* : grâce à la définition d'un fichier ADL en XML pour l'architecture et d'une implémentation en langage impératif, notamment Java.
- *Des ports de communication explicitement liés* : tous les composants doivent avoir des ports de communication bien distincts, reliés explicitement avec des liaisons qui répondent à des protocoles hétérogènes.
- *Des composants faiblement couplés* : les liaisons entre les différents composants peuvent être établies de différentes manières, indépendamment du code du composant.

– *Liaisons uni-directionnelles* : on doit distinguer entre une requête et sa réponse, qui peuvent avoir des contraintes de sécurité différentes.

L'architecture de CIF est faite d'une manière **modulaire**, ce qui le rend **extensible** pour supporter plusieurs modèles de composants et leurs ADL, ainsi que pour intégrer de nouveaux types de modèles d'étiquettes.

CIF dispose principalement de deux outils : l'outil **CIFIntra** pour la vérification du code de chaque composant et **CIFInter** pour la vérification et la sécurisation des liaisons entre les composants. Ces deux outils ont besoin de manipuler des données XML à partir du fichier ADL et du fichier Policy. Pour faciliter leur extraction et utilisation, nous définissons l'API **CIForm**.

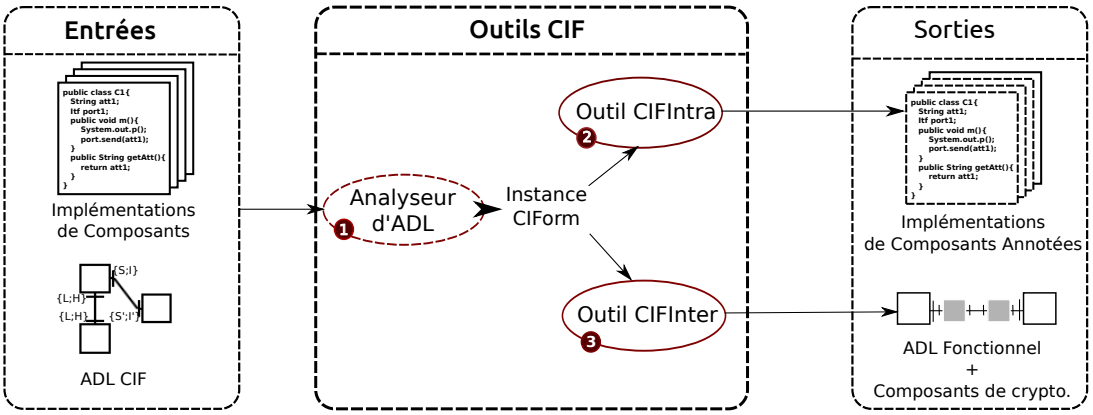


Figure III.I.2 – Étapes de compilation d'un système avec CIF

La figure III.I.2 montre les différentes étapes suivies par les outils CIF pour vérifier la non-interférence d'un système distribué à base de composants. Comme entrée, les outils CIF disposent d'un ensemble de fichiers sources contenant le code d'implémentation des différents composants, ainsi que d'un ensemble de fichiers XML (appelés ici *ADL CIF*) contenant le (ou les) fichiers ADL de description d'architecture ainsi que le fichier *Policy* décrivant la politique de sécurité.

Les outils CIF commencent par analyser les fichiers ADL grâce au module *Analyseur d'ADL* qui va générer une instance CIForm. Cette instance sera ensuite transmise aux moteurs CIFIntra et CIFInter. CIFIntra génère un ensemble de fichiers Java annotés avec des étiquettes de sécurité, et CIFInter génère de nouveaux fichiers ADL contenant les composants cryptographiques insérés entre les composants fonctionnels. Ces différentes étapes seront explicitées dans les parties qui suivent.

2.1 CIForm : CIF Intermediate Format

CIForm (*CIF Intermediate Format*) est un modèle objet (ou API pour *Application Programming Interface*) que nous avons créé pour faciliter la manipulation de l'information donnée dans les fichiers de description d'architecture et de la configuration de sécurité (ADL et *Policy*). La figure III.I.3 représente l'architecture du générateur.

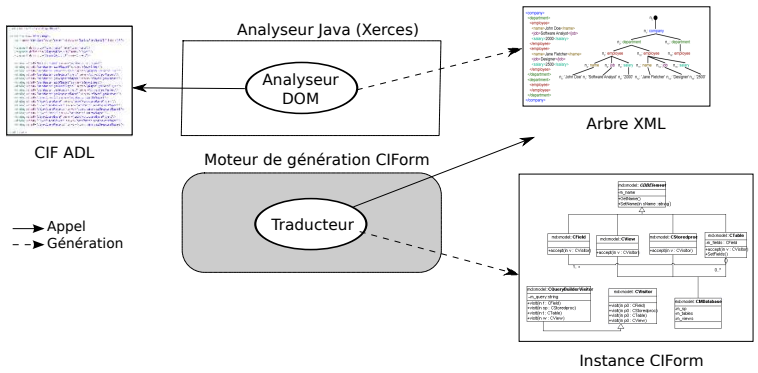


Figure III.I.3 – Outil de génération CIForm

Remarque. Dans cette figure (ainsi que toutes les figures des outils CIF), les blocs grisés représentent des modules que nous avons réalisés et les blocs blancs des modules existants.

L'API CIForm définit un ensemble de classes Java qui décrivent les éléments nécessaires pour la description d'un système à base de composants sécurisé. Il permet de sauvegarder l'ensemble des composants, leurs liaisons, leurs attributs, les protocoles de communication et les politiques de sécurité de manière indépendante du modèle à base de composants utilisé par le concepteur de l'application. En effet, il existe une seule API CIForm, alors que les modèles orientés composants sont nombreux. Par contre, nous définissons un traducteur CIForm pour chaque modèle. Dans notre prototype, nous avons défini un traducteur CIForm pour Fractal, et un traducteur CIForm pour SCA. Le *Moteur de génération CIForm* peut être étendu pour définir d'autres traducteurs, dans le cas où le concepteur de sécurité désire utiliser un nouveau modèle orienté composants. Le fait que CIForm soit unique permet de réaliser facilement cette extension, sans altérer le fonctionnement des autres modules de CIF.

Le générateur CIForm analyse le fichier ADL et le fichier Policy (que nous représentons par le terme *ADL CIF*) donnés par l'utilisateur et extrait toutes les informations sur l'architecture et la configuration de sécurité du système orienté composants. Le générateur utilise un analyseur *Xerces pour Java* pour créer l'arbre XML, et un traducteur qui le parcourt et extrait toutes les informations sur la hiérarchie des composants, les ports, attributs, fichiers sources et étiquettes. Une instance CIForm est alors créée, stockant récursivement tous les composants

en commençant par l'élément composite initial. Le programmeur peut alors extraire toutes les informations nécessaires à partir des objets créés en utilisant les méthodes fournies dans les classes CIForm.

La classe la plus importante dans CIForm est la classe **Label**. C'est une classe qui définit le format de l'étiquette de sécurité, et qui dépend du modèle d'étiquettes utilisé. Dans notre prototype, nous avons implémenté cette classe pour deux modèles d'étiquettes : le modèle d'étiquette décentralisé présenté dans la section III.2.1, partie I et le modèle d'étiquettes à base de jetons centralisé présenté dans la section III.2.2, partie I. Dans le reste du code, cette classe est utilisée d'une manière unifiée, quel que soit le modèle d'étiquettes utilisé : pour déclarer une nouvelle étiquette, nous utilisons un constructeur qui prend en paramètre la chaîne de caractères représentant l'étiquette comme elle est définie dans le fichier *Policy*, et pour le suivi du flux d'information, une seule méthode est utilisée : $label_1.leq(label_2)$. Cette méthode *leq* (pour *less or equal than*) retourne *vrai* si $label_1 \subseteq label_2$ et *faux* sinon. Ainsi, si l'administrateur désire utiliser un nouveau modèle d'étiquettes, il lui suffit d'étendre la classe *Label* et de réécrire la méthode *leq*, et tout le reste du code peut rester identique.

En lançant le moteur de génération de CIForm, une instance CIForm est créée, représentant l'architecture de l'application et la politique de sécurité définis. Cette instance sera utilisée conjointement par les moteurs de vérification inter- et intra-composant définis dans ce qui suit.

2.2 Vérification intra-composant

2.2.1 Vérification par propagation d'étiquette : CIFIntra

L'outil CIFIntra (*CIF for Intra-component verification*) parcourt le code pour chercher des flux d'information illégaux. Il utilise le compilateur *Polyglot* [Nystrom03] pour analyser le code Java initial et propager les étiquettes à travers les instructions. Polyglot est un compilateur extensible visant à créer des extensions de langage sans duplication de code et à créer des compilateurs pour des langages similaires à Java. La figure III.I.4 représente l'architecture de l'outil CIFIntra.

Nous utilisons le générateur d'AST (arbre de syntaxe abstraite ou *Abstract Syntax Tree*) de Polyglot pour analyser le code Java d'un composant (étape 1 dans la figure 3) et générer un AST (étape 2) qui sera transmis au moteur CIFIntra.

CIFIntra extrait les étiquettes à partir des objets CIForm (étape 3) grâce à l'unité d'extraction d'étiquettes. Dans l'implémentation, un composant est représenté par une classe Java, les attributs et ports clients sont représentés par des attributs Java et les ports serveurs par des méthodes Java. Une fois ces attributs et méthodes identifiés dans le code, leurs étiquettes leur sont affectées par l'extracteur d'étiquettes et sont considérées comme **immuables**, contrairement

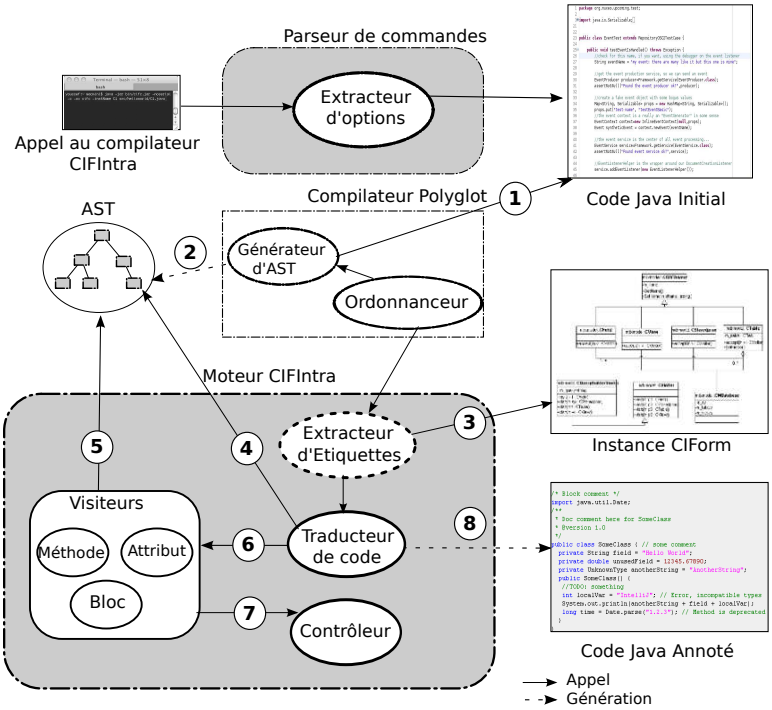


Figure III.I.4 – Architecture du générateur CIFIntra

aux étiquettes **générées**, qui sont affectées aux variables intermédiaires selon leur utilisation dans le code et qui peuvent être modifiées selon l'algorithme présenté ci-dessous.

2.2.2 Comportement du compilateur

On définit la fonction *label* qui, appliquée à une variable, retourne son étiquette et appliquée à une expression, renvoie l'union des étiquettes de toutes ses variables. Le niveau de sécurité du contexte en cours est représenté par l'étiquette *pc* (compteur programme).

Cas d'un flux explicite Pour chaque affectation de la forme $x := y$ (y étant une variable ou une expression), nous devons vérifier que $label(y) \subseteq label(x)$. Dans le cas contraire, et si $label(x)$ n'est pas immuable, x sera alors surclassée, pour avoir la nouvelle étiquette :

$$label(x) := label(x) \bigcup label(y) \bigcup pc$$

Par contre, si $label(x)$ est immuable, deux contraintes doivent être satisfaites :

1. $label(y) \subseteq label(x)$
2. $pc \subseteq label(x)$

Dans le cas particulier où x est un attribut d'objet de la forme $o.f$, il existe un risque de fuite d'information suite au référencement, comme expliqué dans la section III.1.2, partie I. Cela implique qu'une contrainte supplémentaire doit être vérifiée :

3. $label(o) \subseteq label(f)$

Si l'une de ces conditions n'est pas vérifiée, le compilateur déclare alors la présence d'une interférence.

Dans le cas d'un appel de méthode de la forme *o.m(effs)*, avec *effs* un ensemble de paramètres effectifs, nous considérons qu'il existe un flux explicite allant des paramètres effectifs *effs* vers les paramètres formels *forms* déclarés dans la méthode. Chaque paramètre formel doit être au moins aussi restrictif que le paramètre effectif qui lui est associé. Si la méthode est définie comme ayant une étiquette immuable, alors un appel de m avec des paramètres effectifs plus restrictifs que les paramètres formels qui leur sont respectivement associés conduit à une interférence.

D'autre part, dans la déclaration de la méthode m, nous devons nous assurer que :

1. Pour toute variable x dont la valeur a été modifiée dans m, $pc \subseteq label(x)$
2. Pour tout attribut $o.f$ dont la valeur a été modifiée dans m, $label(o) \subseteq label(f)$

Cas d'un flux implicite Les flux implicites représentent des fuites indirectes d'information dans le code qui peuvent faire deviner les valeurs des données secrètes à partir des changements des données publiques. Prenons par exemple l'instruction :

$$\text{si } (h == vrai) \text{ alors } l := vrai \text{ sinon } l := faux\,;$$

Il est clair que cette instruction équivaut à une affectation $l := h$. Ainsi, dans le cas où $label(h) \not\subseteq label(l)$, ce type d'instruction provoque une interférence. Ce type de flux peut être détecté dans le cas où la valeur d'une variable est modifiée dans un bloc dont le contexte est plus restrictif.

Pour traiter les flux implicites, le système maintient le *pc* à jour. Il l'initialise d'abord à $\bot$ (représentant le niveau de sécurité le plus bas). Pour chaque nouveau bloc de contrôle de la forme : (*si (e) alors* S_1 *sinon* S_2) ou (*tant que (e) répéter* S_1), avec e une expression et S_1 et S_2 deux ensembles d'instructions, les actions suivantes sont réalisées :

1. $ancienPc = pc\,;\ pc = pc \bigcup label(e)$
2. Récursivement, l'algorithme est déroulé sur S_1 et S_2 en prenant en considération le nouveau *pc*
3. $pc = ancienPc$

Dans le cas de blocs imbriqués, la valeur du compteur programme *pc* change au niveau de chaque bloc pour réunir la valeur de l'ancien *pc* et l'étiquette de l'expression conditionnelle du bloc. Le *pc* est remis à sa valeur initiale à la sortie de chaque bloc.

Cas d'un flux dû à une exception déclarée Les exceptions en Java peuvent être considérées comme des cas particuliers de sorties. Dans notre algorithme, nous traitons les cas des exceptions récupérées dans un bloc *try/catch*. Dans les autres cas d'exceptions déclarées, nous considérons que chaque composant a un port virtuel supplémentaire noté P_{exp} et étiqueté $\perp$, par lequel s'acheminent toutes les exceptions non récupérées. Pour vérifier la non-interférence, CIFIntra vérifie qu'aucune information secrète n'est divulguée par le port P_{exp}, directement ou indirectement.

Ainsi, chaque lancement explicite d'une exception qui n'est pas traité par un bloc *catch*, avec l'instruction $throw(exp)$ ou bien avec l'appel d'une méthode qui lève une exception, doit être exécuté dans un contexte de bas niveau, c'est-à-dire qu'une valeur du *pc* plus restrictive que $\perp$ produit une interférence.

Dans le cas des exceptions non déclarées (*unchecked exceptions*) et qui ne sont pas traitées dans un bloc *catch*, nous considérons les exceptions les plus usuelles. Ainsi, dans chacun des cas suivants, CIFIntra déclare une interférence :

1. A l'accès à un élément d'un tableau de la forme $t[i]$ et si $label(t) \not\subseteq label(P_{exp})$ ou $label(i) \not\subseteq label(P_{exp})$: risque d'une *ArrayIndexOutOfBoundsException*
2. A l'accès à un objet de la forme $o.f$ ou $o.m(...)$ et si $label(o) \not\subseteq label(P_{exp})$: risque d'une *NullPointerException*
3. A la division par une variable de la forme x/y, ou même directement par un zéro et si $label(x) \not\subseteq label(P_{exp})$ ou si $label(y) \not\subseteq label(P_{exp})$: risque d'une *ArithmeticException.*
4. Au transtypage (*casting*) explicite d'un objet de la forme $(C)o$ et si $label(o) \not\subseteq label(P_{exp})$: risque d'une *ClassCastException.*

Dans le cas d'un bloc de la forme (try $\{S_1\}$ catch(exp) $\{S_2\}$ finally $\{S_3\}$) :

1. Si S_1 contient une instruction $throw(exp)$ ou appelle une méthode qui déclare une exception avec l'instruction $throws(exp)$, dans un contexte pc_1, alors dans le bloc *catch* qui lui est associé, la valeur du $pc = pc_1$.
2. Pour le bloc *finally*, étant donné qu'il est exécuté dans le cas d'une exception ou dans le cas d'une terminaison normale, son *pc* est la réunion de l'ancien *pc* et de la réunion des étiquettes des variables qui risquent de produire des exceptions.

Le compilateur procède à la vérification des instructions une à une de manière séquentielle. Il peut modifier le niveau de sécurité d'une variable dont l'étiquette est générée, si elle est utilisée dans un contexte plus restrictif. A chaque changement d'étiquette, le compilateur vérifie que ce changement ne provoque pas de flux illégaux dans les instructions qui le précèdent. Il revérifie donc le code du composant à chaque fois qu'une étiquette est modifiée.

2.2.3 Utilisation du module Contrôleur

Le comportement du compilateur peut s'avérer dans certains cas assez restrictif. Cela est dû au fait qu'il est impossible de trouver des systèmes réels sans aucune interférence entre les données privées et les données publiques. C'est pour cela qu'une interférence qui est détectée par le compilateur CIFIntra peut s'avérer utile voire obligatoire pour le bon fonctionnement du système. Un exemple très connu d'interférence est celui d'une opération d'authentification. Une interférence est facilement détectée entre la donnée privée (le mot de passe) et la donnée publique (authentification réussie ou non) dont la valeur dépend de celle du mot de passe. Ce type d'interférences doit être toléré pour que le système puisse fonctionner. Une amélioration de l'algorithme en tenant compte de la propriété de sensibilité au flux (*flow sensitivity*) [Amtoft06, Hunt06, Russo10] pourrait relâcher certaines contraintes.

Pour éviter ce problème, nous utilisons la notion de rétrogradation [Sabelfeld09]. Pour l'appliquer, un module **Contrôleur** est appelé à chaque fois qu'une interférence est détectée(voir la figure III.I.4). Ce module vérifie d'abord s'il peut résoudre l'interférence à son niveau sans intervention de l'utilisateur. Cela implique rétrograder le niveau de sécurité de la (ou les) variable(s) responsable(s) de l'interférence. La capacité du contrôleur à rétrograder le niveau de sécurité d'une donnée dépend des **capacités** (*capabilities*) de chaque composant, définies au niveau du fichier *Policy*, comme indiqué dans la section I.1.1, partie III. Une capacité représente la possibilité qu'a un composant d'ajouter ou supprimer des droits. Chaque composant a un ensemble de capacités, qui définissent son aptitude à rétrograder une information. L'expression des capacités dépend du modèle d'étiquettes utilisé. Par exemple, si nous utilisons le modèle d'étiquettes à base de jeton (section III.2.2, partie I), $t+$ représente la capacité d'un composant à ajouter un jeton t à ses étiquettes.

Si la capacité du composant indique qu'il a la possibilité de dégrader le niveau de la donnée responsable de l'interférence, le contrôleur autorise l'interférence et demande au compilateur de continuer son travail. Sinon, le contrôleur affiche un message pour indiquer à l'administrateur l'endroit où l'interférence a été détectée. L'administrateur du système a ainsi la possibilité d'autoriser ou d'interdire l'interférence.

2.2.4 Génération du code annoté

Le **traducteur de code** (étape 8) génère un fichier Java annoté (à la manière d'un fichier *JIF*) que l'administrateur pourra consulter pour vérifier le résultat de la propagation des étiquettes. Le but principal de ce fichier est de permettre le débogage du compilateur CIFIntra. Il n'est pas compilable.

2.2.5 Gestion des composants patrimoniaux par construction de liaisons internes

La gestion des composants patrimoniaux est une nouveauté proposée par notre plateforme pour autoriser les concepteurs à utiliser des boîtes noires dans leurs systèmes, sans craindre qu'elles n'enfreignent la politique de sécurité désirée par le concepteur.

Chaque composant patrimonial devra être accompagné d'un document représentant ses **liaisons internes**. Ces liaisons représentent les relations entre les entrées du composant (les ports serveur et les attributs) et ses sorties (les ports clients). Cela nous permettra de représenter le système sous la forme d'un ensemble de liaisons, qu'il suffira de suivre pour voir l'évolution du flux d'information. La figure III.I.5 illustre un exemple de système avec les liaisons internes et externes.

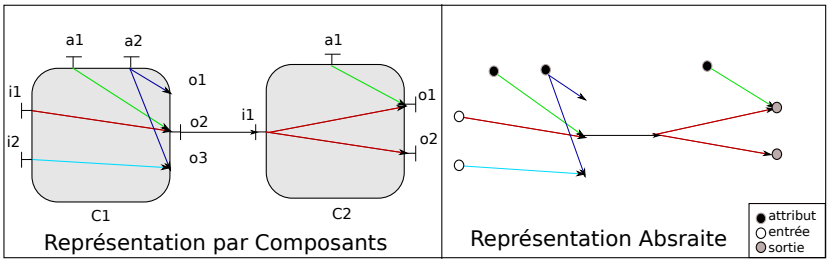


Figure III.I.5 – Liaisons internes

Une liaison est établie entre une entrée et une sortie d'un composant si et seulement s'il existe un flux d'information allant de l'entrée vers la sortie. Un flux de données est représenté par la fonction *depends(o,i)* qui indique que la sortie o dépend de l'entrée i.

Nous avons réalisé un outil de construction des liaisons internes, que nous avons appelé **IBP** (pour *Internal Binding Plotter*). Grâce à cet outil, le développeur du composant patrimonial n'a pas besoin de dévoiler le code de son composant au concepteur du système cible : il peut juste fournir une attestation certifiée montrant les liaisons internes de son composant, appelée **IBA** (pour *Internal Binding Artifact*). Le concepteur du système cible appliquera les étiquettes aux ports d'entrée et de sortie, et l'outil CIFIntra procédera à la vérification de la non-interférence, en utilisant les liaisons internes, de la manière suivante :

– Soit o un port de sortie du composant. Le port o est relié, selon son IBA, au port d'entrée i et à l'attribut a. Cela veut dire qu'il existe un flux d'information allant de i vers o et de a vers o.
– Pour que la sortie o soit non-interférente, il faut que

$$L(i) \subseteq L(o) \quad et \quad L(a) \subseteq L(o)$$

IBP est basé sur le compilateur Polyglot. Il utilise les visiteurs pour parcourir l'arbre de syntaxe abstraite, et chercher les dépendances entre les différentes variables.

Les variables sont divisées en trois types différents :
– *Entrées* : représentant les ports d'entrée du composant ainsi que ses attributs
– *Sorties* : représentant les ports de sortie du composant
– *Variables intermédiaires* : représentant les autres champs ainsi que les variables locales de la classe.

L'outil prend en entrée le fichier ADL contenant la liste des entrées et sorties, ainsi qu'un fichier Java contenant le code du composant. Il parcourt ensuite le code instruction par instruction, et construit les dépendances entre les variables. Les conditions de dépendance sont représentées par l'annexe A, partie IV.

Une fois toutes les dépendances construites, l'outil les parcourt de nouveau de manière récursive pour éliminer les variables intermédiaires et ne laisser que les variables immuables, soit les entrées et les sorties des composants.

2.3 Vérification inter-composants

L'outil CIFInter(*CIF for Inter-component verification*), dont l'architecture est décrite dans la figure III.I.6, réalise deux tâches : (1) vérifier pour chaque liaison si la donnée circule dans la bonne direction, c'est à dire, du port le moins restrictif vers le plus restrictif et (2) mettre en œuvre la confidentialité et l'intégrité des données envoyées en générant automatiquement des composants cryptographiques entre les composants fonctionnels.

2.3.1 Contrôle de flux d'information pour les liaisons

La première étape est réalisée par le **vérificateur de flux d'information**. Cette unité parcourt les objets CIForm et vérifie pour chaque liaison que l'étiquette du port client est moins restrictive que celle du port serveur. Cela est fait en faisant appel à la méthode *leq* définie dans la classe *Label* de CIForm dans le but de comparer les étiquettes.

Figure III.I.6 – Architecture du générateur CIFInter

Si $label(p_{client}).leq(label(p_{serv}))$, alors le flux d'information ne présente pas d'interférence. Sinon, une exception de type *LabelException* est lancée, indiquant à l'utilisateur la liaison qui génère l'erreur. Si la vérification se termine sans exception, le vérificateur passe la main aux **générateurs fonctionnels**.

2.3.2 Insertion des composants cryptographiques

L'objectif des générateurs est de modifier le fichier qui décrit l'architecture du système en y insérant des composants cryptographiques entre les composants fonctionnels. Ces composants se chargent du chiffrement et de la signature des messages échangés. Pour l'instant, nous utilisons l'algorithme RSA pour le chiffrement et MD5/RSA pour la signature, mais dans des travaux futurs, le choix des algorithmes pourra être configurable dans le fichier Policy.

Nous avons réalisé les transformations CIF-Fractal et CIF-SCA, mais les générateurs peuvent être étendus à d'autres langages orientés composants, basés sur la séparation de l'architecture et du comportement.

Nous définissons un composant cryptographique par composant fonctionnel. Ce composant va intercepter toutes les communications sortantes et entrantes du composant fonctionnel associé qui nécessitent un chiffrement ou une signature. Le code de chaque composant cryptographique est généré automatiquement par le compilateur CIFInter. Ce compilateur se base sur un modèle de classe que nous avons prédéfini, qu'il va modifier et adapter selon les différentes interfaces nécessaires pour chaque composant et selon les opérations de cryptographie qu'il doit appliquer dessus. Nous présentons ce modèle (appelé *SecurityImplTemplate*) dans l'annexe B, partie IV.

L'annexe D, partie IV présente les primitives cryptographiques fournies par le langage Java, qui ont été utilisées dans notre implémentation.

Conclusion

Nous avons présenté dans ce chapitre la contribution principale de notre travail, soit la vérification de la non-interférence à la compilation pour les systèmes distribués à base de composants. Nous avons montré la chaîne d'outils CIF qui permet d'automatiser la vérification de la non-interférence en partant d'une spécification de haut niveau, et de générer les primitives cryptographiques nécessaires pour le transport des messages dans un réseau non sécurisé. Nous avons également présenté notre solution pour les composants patrimoniaux, qui consiste en la fourniture des différentes liaisons internes des composants hérités, sans avoir besoin de divulguer leurs codes.

Cependant, les outils CIF permettent de réaliser des systèmes sécurisés par construction, mais ne garantissent pas la préservation de la politique de sécurité pendant l'exécution, si le système subit des changements au niveau de son architecture. Nous proposons dans le chapitre suivant une plateforme pour l'application dynamique de la non-interférence à des systèmes à base de composants déjà sécurisés avec CIF.

Chapitre II

DCIF : Modèle de vérification dynamique de la non-interférence

DCIF (*Dynamic Component Information Flow*) est un canevas orienté composants pour la construction de systèmes distribués sécurisés à la compilation et à l'exécution. Il définit deux types de composants : des **composants fonctionnels** qui représentent l'application distribuée et les **composants de gestion**, qui permettent d'assurer des propriétés non fonctionnelles, en particulier la sécurité.

Nous définissons un ensemble de termes que nous utiliserons dans la suite :

- *Nœud* : un nœud représente une machine. Les connexions entre des composants d'un même nœud sont sûres, il n'ont donc pas besoin de cryptographie.
- *Domaine* : un domaine est un ensemble de composants qui se font mutuellement confiance, et qui communiquent via un réseau sûr, qui n'a donc pas besoin de mécanisme pour sécuriser les données échangées. Un domaine contient un ensemble de nœuds.

1 Architecture de DCIF

La figure III.II.1 représente l'architecture de DCIF. Nous détaillons dans ce qui suit les différents composants de sécurité que nous avons défini.

Global Manager (GM) : Le gestionnaire global est un composite qui regroupe l'ensemble des composants non fonctionnels, dont les composants de sécurité. À la base, il existe un seul GM partagé et centralisé, mais pour certaines applications, il peut être dupliqué sur plusieurs nœuds pour des besoins de disponibilité et de sécurité.

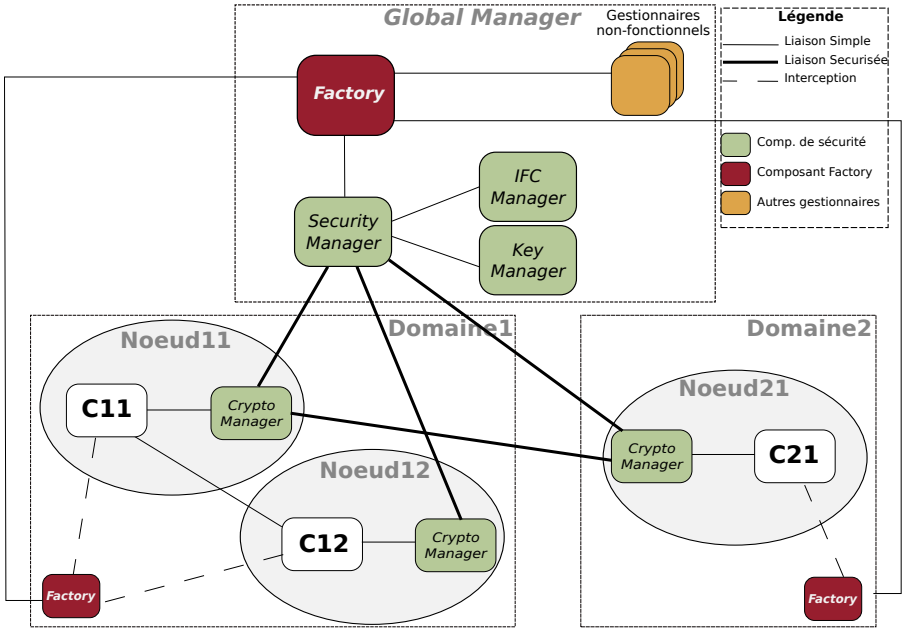


Figure III.II.1 – Architecture de DCIF

Factory : Le Factory (*l'usine*) est un composant responsable de toutes les opérations de mise à jour de l'architecture du système. Il existe deux types de composants Factory : **local** et **global**. Le Factory local est propre à chaque domaine, et intercepte les opérations de mise à jour des composants du domaine. Il les transmet ensuite au Factory global, qui prend les décisions adéquates.

Key Manager(KM) : Le gestionnaire de clefs est le composant responsable de la sauvegarde et de la gestion de l'ensemble des clefs cryptographiques du système. Chaque domaine et chaque composant fonctionnel ont une clef cryptographique distincte, toutes stockées dans le KM.

Information Flow Control Manager (IFCM) : Le gestionnaire de flux d'information est le composant responsable du contrôle de flux d'information du système. Il stocke l'ensemble des étiquettes de sécurité des différents ports et attributs, leurs capacités, ainsi que les certificats IBA des liaisons internes générés à la compilation pour chaque composant patrimonial du système (voir section I.2.2.5, partie III). Il se charge de la vérification de la propriété de non-interférence pour chaque opération de mise à jour dynamique.

L'IFCM permet également de prendre des décisions de **rétrogradation** des informations. Ces décisions sont basées sur deux concepts :

– *Capacités* : ce sont les capacités liées à chaque composant, comme expliqué dans la section I.2.2.3, partie III. Il définissent la possibilité d'un composant à rétrograder le niveau de sécurité d'une information.

– *Liste de confiance* : Cette liste représente, pour chaque composant, l'ensemble des composants en qui il a confiance. Cette liste est maintenue par l'IFCM pour valider l'établissement automatique des liaisons entre composants de confiance.

Crypto Manager (CM) : le gestionnaire des opérations cryptographiques est le composant responsable du chiffrement et de la signature des messages à destination de composants dans des domaines différents. Il est propre à chaque nœud.

Security Manager (SM) : Le gestionnaire de sécurité est le composant principal de gestion de la sécurité du canevas. Il se charge de tous les transferts entre les composants KM, IFCM, CM et Factory.

La figure III.II.2 détaille l'architecture interne du composant IFCM.

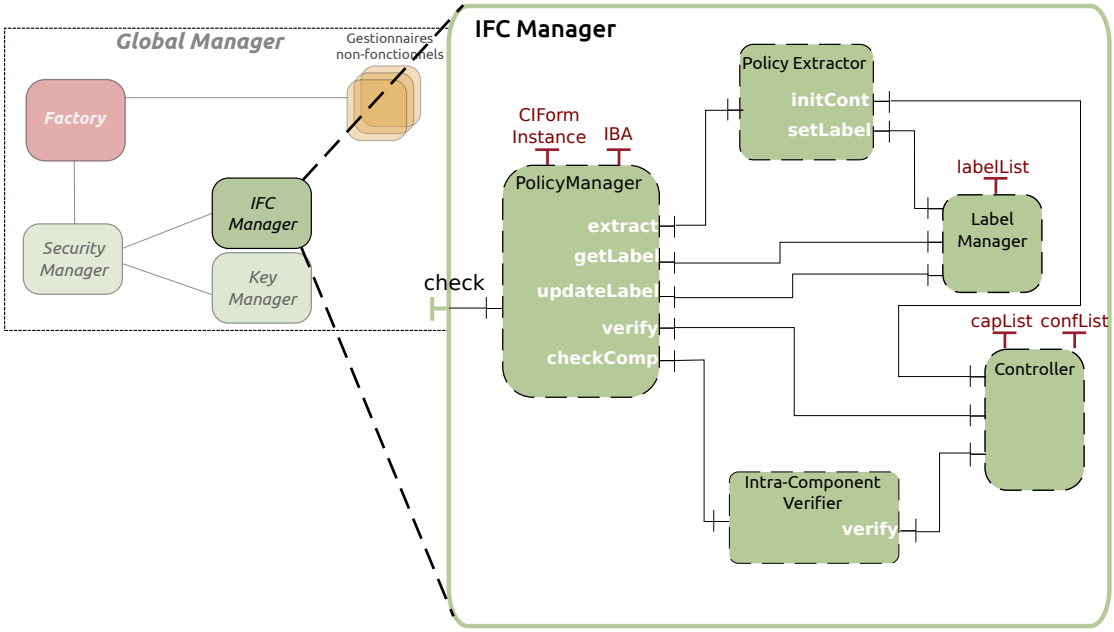


Figure III.II.2 – Contenu du composant de gestion des flux d'information

L'IFCM contient cinq sous-composants :

Policy Manager : C'est le sous-composant principal qui orchestre les communications entre les autres sous-composants. Il permet de réaliser les opérations principales de vérification de flux. Il définit une interface serveur *check* reliée au Security Manager et cinq interfaces clientes :

– **extract** : Cette interface est chargée de l'extraction des informations à partir des fichiers *Policy* fournis par le système. Le résultat retourné est une instance de CIForm stockée dans l'attribut *CIForm instance*.

– **getLabel** : Cette interface permet de retourner l'étiquette d'une entité donnée.

- **updateLabel** : Cette interface permet de modifier l'étiquette d'une entité.
- **verify** : Cette interface permet de vérifier si une interférence est autorisée ou pas en consultant les listes des capacités et des composants de confiance via le contrôleur.
- **checkComp** : Cette interface permet de vérifier la propriété de la non-interférence dans le code d'un composant donné.

Le Policy Manager stocke également tous les certificats IBA des composants patrimoniaux du système.

Policy Extractor : Ce composant est responsable de l'extraction des informations à partir des fichiers *Policy* fournis par le système. Il envoie ensuite les étiquettes extraites grâce à son interface **setLabel** et les capacités via l'interface **setCap**, puis retourne au Policy Manager une instance de CIForm contenant l'architecture du système.

Label Manager : Ce composant associe à chaque élément du système cible son étiquette. Il les stocke dans l'attribut *labelList*.

Controller : Ce composant permet de décider si une interférence est autorisée ou non en consultant la liste des capacités (**capList**) et la liste de confiance (**confList**) du composant responsable de l'interférence.

Intra Component Verifier : Ce composant réalise la vérification intra-processus décrite dans la section I.2.2, partie III. Il est appelé au déploiement, ainsi qu'à l'arrivée d'un nouveau processus. Si ce composant détecte une interférence, il envoie une demande de vérification au contrôleur via son interface **verify**.

2 Mise en place du canevas et scénario d'exécution

2.1 Mise en place du canevas

On définit une clef symétrique par domaine et une clef symétrique par composant fonctionnel. Au démarrage, le KM contient l'ensemble des clefs de tous les domaines, ainsi que de tous les composants. Chaque composant CM d'un nœud N stocke la clef du domaine auquel il appartient, ainsi que toutes les clefs des composants du nœud N.

Toute liaison entre un composant C_1 d'un domaine D_1 et un composant C_2 d'un domaine D_2 doit passer par les CM des nœuds N_1 et N_2 auxquels appartiennent C_1 et C_2.

2.2 Envoi et réception de messages

Pour envoyer un message d'un composant C_1 à un composant C_2 du même domaine, aucune protection n'est nécessaire : le message est envoyé directement. Mais si les deux composants

sont dans des domaines différents, le message doit être protégé contre d'éventuelles attaques externes du réseau. Il doit donc passer par les composants de cryptage CM_1 et CM_2.

CM_1 (le *Crypto Manager* du nœud N_1 de C_1) vérifie s'il a déjà établi une liaison sécurisée avec C_2. Si c'est le cas, cela veut dire que la clef privée commune à C_1 et C_2 est stockée dans CM_1. Le message est donc chiffré avec cette clef, et envoyé à CM_2, qui va le déchiffrer et le transférer à C_2.

Si cette connexion est la première entre C_1 et C_2, CM_1 consulte le SM pour avoir la clef symétrique de C_2. SM transmet la requête à KM, qui va générer une clef privée pour la connexion entre C_1 et C_2 et la renvoie au SM. Cette clef sera envoyée à CM_1 et CM_2 pour pouvoir autoriser une nouvelle communication entre C_1 et C_2. SM chiffre la clef de session avec la clef du domaine D_1 (resp. D_2) et l'envoie à CM_1 (resp. CM_2), qui va la déchiffrer, et l'utiliser pour le chiffrement et la signature (respectivement déchiffrement et vérification) du message à envoyer. CM_1 envoie donc le message chiffré et signé à CM_2. À la réception du message, CM_2 le vérifie et le déchiffre, puis le transmet à C_2.

2.3 Reconfiguration dynamique

Ajout d'un composant Quand un nouveau composant C d'un nœud N et d'un domaine D va être ajouté au système, le Factory global vérifie s'il appartient à un nœud déjà défini dans le système. Si le nœud n'est pas reconnu, le Factory local au domaine D génère un CM pour le nouveau nœud. Une fois le CM défini, une clef symétrique pour C est générée par le KM, et envoyée à CM, chiffrée avec la clef de D.

Avant l'insertion du composant dans le système, le Factory envoie au IFCM les informations sur ce composant pour qu'il procède à une vérification intra-composant. Le IFCM envoie les fichiers *Policy* au composant *Policy Extractor*, qui va stocker les étiquettes du nouveau composant dans le *Label Manager* et ses capacités dans le *Controller*, puis procède à une vérification du code du composant en appelant le composant *Intra-Component Verifier*. Ce dernier procède à la propagation des étiquettes dans le code du composant. Dans le cas où le composant est patrimonial, un certificat IBA est fourni avec le fichier *Policy* et est envoyé au *Intra-Component Verifier* qui va vérifier si le composant satisfait la politique de sécurité désirée. Si une interférence est détectée, le composant Contrôleur est appelé pour vérifier, selon les capacités des composants, si elle est autorisée ou pas.

Suppression d'un composant Quand un composant C d'un nœud N et domaine D est supprimé, le Factory envoie au SM pour l'en informer. Le SM transfert la requête au KM, qui va supprimer la clef de C de la base de données.

De plus, le IFCM supprime les étiquettes relatives à ce composant en appelant l'interface *updateLabel* du composant *Policy Manager*. Ce dernier modifie également l'instance CIForm pour mettre à jour l'architecture du système.

Remplacement d'un composant Le remplacement d'un composant par un autre peut être réalisé suite, par exemple, à la détection d'un composant fautif par le *Fault Manager*, composant non-fonctionnel défini dans le GM [1]. Quand le composant fautif est identifié, une demande est envoyée au Factory pour qu'il réalise son remplacement.

Remplacer un composant équivaut à deux opérations d'ajout et de suppression successives décrites plus haut.

Migration d'un composant Quand un composant migre d'un nœud à un autre, cela équivaut à une opération de suppression du composant du nœud source suivie d'une opération d'ajout de ce composant dans le nœud destination.

Ajout d'une liaison Quand une liaison doit être établie dynamiquement entre un composant $C_1(N_1,D_1)$ et un composant $C_2(N_2,D_2)$, le composant Factory envoie au composant SM les détails de la requête, en associant à chaque interface son étiquette de sécurité qu'il extrait du fichier *Policy*.

Les liaisons peuvent avoir deux états : *établie* si la restriction de non-interférence est respectée, ou *en attente* sinon. Le IFCM implémente la politique de sécurité du système : il peut soit décider de rétrograder certaines étiquettes de sécurité, soit demander au composant Factory de reconfigurer l'architecture de manière à ce que les liaisons qui violent la propriété de non-interférence soient mises *en attente*.

Le IFCM vérifie si la liaison demandée respecte la règle de non-interférence, à savoir que $\ell(client) \subseteq \ell(serveur)$. Pour cela, il extrait les deux étiquettes des extrémités de la liaison grâce à l'interface *getLabel* du *Policy Manager*. Ce dernier les compare : Si aucune interférence n'est détectée, l'interface serveur *check* retourne *true*, ce qui implique que la liaison peut être établie. Si la liaison n'est pas autorisée, le *Policy Manager* procède à l'appel du contrôleur via l'interface *verify*. Le contrôleur consulte d'abord la liste des capacités de C_1 : est-ce que C_1 est en mesure de rétrograder l'étiquette affectée à son port ? Si ce n'est pas le cas, il consulte en deuxième lieu la liste de confiance : est-ce que C_1 fait confiance à C_2 ? Si c'est le cas, une liaison peut être établie même si elle ne respecte pas la propriété de non-interférence.

1. L'implémentation du composant *Fault Manager* ainsi que de tous les autres composants non-fonctionnels est hors de propos dans notre travail.

Si le IFCM autorise la liaison, le composant Factory procède à son établissement. Si C_1 et C_2 sont dans le même domaine, une liaison directe est établie entre C_1 et C_2. Sinon, le Factory intercale les composants CM_1 et CM_2 entre C_1 et C_2 pour permettre le chiffrement et la signature de la donnée avant son envoi.

Modification d'une étiquette de sécurité La configuration de sécurité initiale peut être modifiée (reconfigurée) grâce à l'aspect dynamique offert par le composant contrôleur. Ainsi, pour chaque reconfiguration des étiquettes, le système doit vérifier de nouveau la validité de la nouvelle configuration de sécurité et mettre à jour l'architecture du système selon cette nouvelle configuration. La modification de l'architecture porte surtout sur les liaisons entre les interfaces. C'est-à-dire, lorsqu'il y a un risque de fuite de données, la liaison entre les deux interfaces concernées est mise en attente. Également, on peut avoir le cas où la liaison était initialement en attente, et suite aux changements des étiquettes, il n'y a plus de risque de fuite ce qui fait que Factory établit la liaison. Ainsi, chaque fois qu'une étiquette est modifiée par l'interface *updateLabel*, le *Policy Manager* procède à la vérification de l'impact de ce changement sur le reste du système en parcourant l'instance CIForm. Ce changement peut donc provoquer des suppressions ou des ajouts de liaisons, qui sont notifiés au Factory.

Mise à jour d'un composant La mise à jour d'un composant existant se manifeste soit par la modification d'une ou plusieurs de ses étiquettes de sécurité (cf. le cas précédent), soit par la modification de son comportement. Dans ce cas, le composant Factory intercepte tout changement dans le code du composant, et envoie une demande de vérification au IFCM, qui fait appel au composant *Intra-component Verifier*. Ce dernier vérifie la propriété de non-interférence dans le code. Toute interférence détectée est envoyée au Contrôleur pour vérification.

Conclusion

Le canevas de vérification dynamique de la non-interférence DCIF permet d'assurer que la politique de sécurité définie à la compilation est préservée à l'exécution également, face à des changements dynamiques de l'architecture du système. Cela est réalisé grâce à un ensemble de composants non fonctionnels qui automatisent le processus de vérification de flux et de génération du code cryptographique de manière transparente à l'utilisateur.

Nous présentons dans la partie suivante une partie d'évaluation, qui comporte un ensemble d'études de cas permettant d'illustrer avec des exemples concrets l'application des solutions montrées dans ce travail, ainsi qu'une évaluation des performances des outils CIF et un début d'étude formelle pour l'outil CIFIntra.

Quatrième partie

Évaluation et Validation

Le vrai génie réside dans l'aptitude à
évaluer l'incertain, le hasardeux,
les informations conflictuelles.

[Winston Churchill]

Chapitre I

Études de Cas

Nous présentons dans ce chapitre plusieurs études de cas que nous avons implémenté. Nous avons choisi des applications qui diffèrent par leurs types et leurs comportements, et avons montré comment est-ce que notre solution propose de résoudre les problèmes posés par chacune d'entre elles :

- **La bataille navale** : cet exemple, initialement écrit en JIF, a été réécrit en Fractal et sécurisé avec CIF. C'est un exemple de référence qui montre l'utilisation du modèle d'étiquettes DLM.
- **La réservation de billet d'avion** : Cet exemple montre l'application de CIF aux systèmes utilisant les services web.
- **La clinique de chirurgie esthétique** : Cette étude de cas a été implémentée pour montrer l'application des outils CIF sur les systèmes à échelle réelle, en utilisant le modèle d'étiquettes à jetons et la spécification SCA.
- **Le calendrier partagé** : Cette application montre l'utilisation de DCIF pour la planification d'une réunion, et son utilité pour protéger la confidentialité des informations pendant l'exécution.

1 Jeu de la bataille navale

Le jeu de la bataille navale est extrait de [Myers06] et initialement développé en JIF. Nous avons reproduit ce système sous forme de composants et réécrit avec le modèle Fractal. Nous utilisons pour cette application le DLM (décrit dans la section III.2.1, partie I) pour représenter les niveaux de sécurité.

Description du système

Le jeu de bataille navale fait intervenir au moins deux joueurs et un coordinateur. Chaque joueur possède une grille (*Board*) secrète contenant un nombre fixé n de bateaux dont les coordonnées sont choisies au début du jeu. Chaque joueur essaie de deviner les coordonnées des bateaux de l'adversaire. Le vainqueur est celui qui aura deviné toutes les coordonnées de son adversaire en premier. Un coordinateur garde une copie des grilles des deux joueurs pour s'assurer qu'aucun des deux n'a modifié la position des bateaux en cours du jeu. Afin de contrôler le jeu, le coordinateur sert d'intermédiaire dans l'envoi des requêtes (coordonnées proposées par les joueurs) et les réponses (variables booléennes disant si les tentatives ont réussi ou échoué).

Nous représentons dans la figure IV.I.1 le système correspondant au scénario décrit ci-dessus.

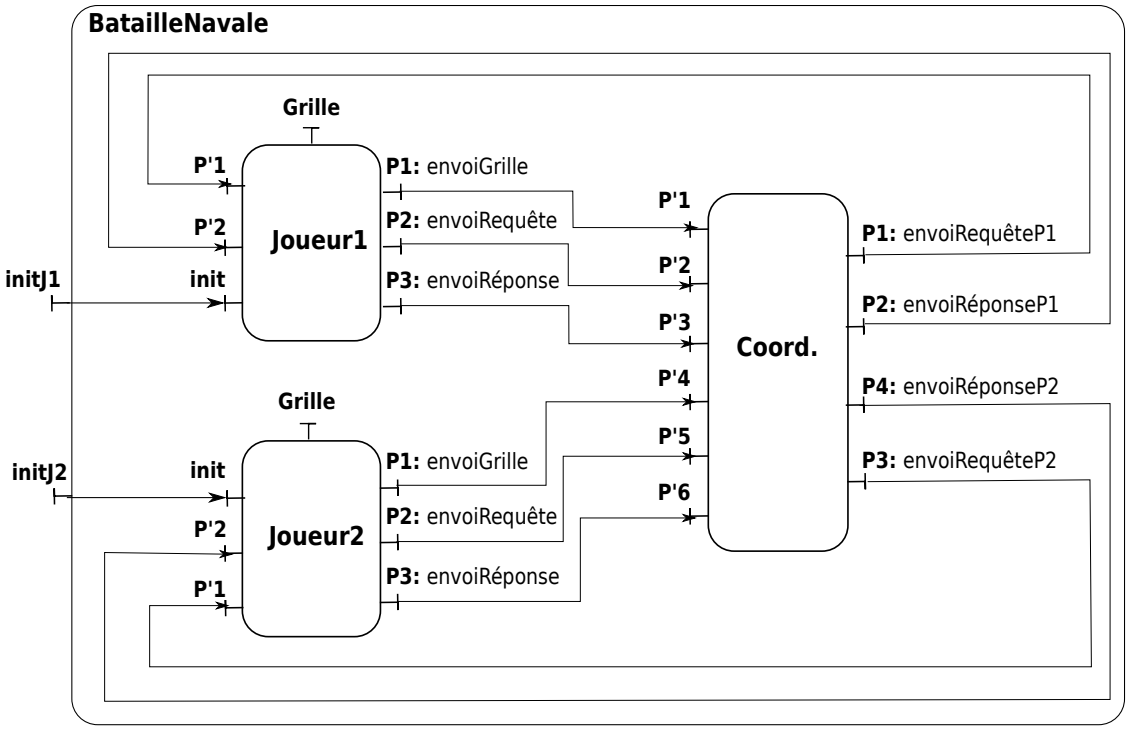


Figure IV.I.1 – Architecture du jeu de la bataille navale

Le protocole de communication se fait de la manière suivante :

- Chaque joueur crée une grille pleine où il dispose ses bateaux à sa guise et une grille vide qu'il utilisera pour enregistrer les réponses de son adversaire. Il envoie ensuite sa grille au coordinateur via le port *P1* (*envoiGrille*).
- Le jeu commence :

- Le premier joueur (*Joueur1*) envoie sa requête via le port *P2* (*envoiRequête*) au coordinateur.
- Le coordinateur transmet la requête au second joueur (*Joueur2*) via le port *P3* (*envoiRequêteP2*).
- Le *Joueur2* exécute la requête, met à jour sa grille et envoie la réponse au coordinateur via le port *P3* (*envoiRéponse*).
- Le coordinateur vérifie la justesse de la réponse, met à jour sa copie de la grille du *Joueur2* et transmet la réponse au *Joueur1* via le port *P2* (*envoiRéponseP1*). Si le joueur 2 a triché, le coordinateur arrête le jeu.
- *Joueur1* reçoit la réponse à sa requête, et met à jour ses données sur sa copie de la grille du *Joueur2*.

– Réciproquement pour le *Joueur2*.
– Le jeu continue ainsi jusqu'à ce que l'un des joueurs trouve tous les bateaux de son adversaire.

Attribution des niveaux de sécurité

Les différentes autorités de notre système sont le *Coordinateur*, *Joueur1* et *Joueur2*. Chacune de ces autorités est représentée par un composant.

Les niveaux de sécurités seront attribués :

1. Aux attributs des composants : Notamment l'attribut *grille* de chacun des composants *Joueur1* et *Joueur2*.
2. Aux ports clients et serveurs des composants, qui servent pour l'envoi des grilles ainsi que des différentes requêtes et réponses tout au long du jeu.

L'attribution des niveaux de sécurité se fait selon l'usage que le concepteur désire accorder à chacune des autorités pour la manipulation des données. Dans cet exemple, nous considérons que :

– Le port *init* attribué à chaque joueur sert à démarrer la partie en envoyant le nombre de bateaux à construire. Ce nombre n'étant pas confidentiel et ne portant l'empreinte d'aucune autorité, son niveau de sécurité est {}.
– Une grille est la propriété du joueur (x) qui l'a créée. Nous supposons que, initialement, aucune autre autorité n'a la possibilité de consulter cette grille. En envoyant la grille, le niveau de confidentialité est alors $(J_x \rightarrow *)$
– Les requêtes et réponses sont les propriétés du joueur (x) qui les initie. Elles peuvent être consultées par le Coordinateur et l'adversaire (3-x). Quand le joueur (x) envoie une

requête ou une réponse au coordinateur via un port client, le niveau de confidentialité est alors $(J_x \rightarrow C, J_{(3-x)})$.

- Du point de vue de l'intégrité, toutes les données envoyées par un joueur (x) sont garanties comme portant son empreinte seulement, personne d'autre ne l'ayant modifié jusque là. Tout envoi partant d'un joueur vers le coordinateur a le niveau d'intégrité $(J_x \leftarrow)$.
- En recevant la grille d'un joueur, le coordinateur désire y appliquer des modifications, en y indiquant, par exemple, quelles sont les coordonnées qui ont été jouées. En recevant la grille sur un port serveur en provenance d'un joueur (x), le niveau d'intégrité est donc $(J_x \leftarrow C)$.
- Quand le coordinateur transmet les requêtes et réponses aux joueurs appropriés, ces données doivent garder le même niveau de sécurité. Un port qui envoie d'une requête ou d'une réponse d'un joueur (x) vers son adversaire (3-x) de la part du coordinateur a le niveau de sécurité $\{J_x \rightarrow C, J_{(3-x)}; Jx \leftarrow \}$.

Ces hypothèses nous permettent de répartir les étiquettes comme représenté dans la figure IV.I.2.

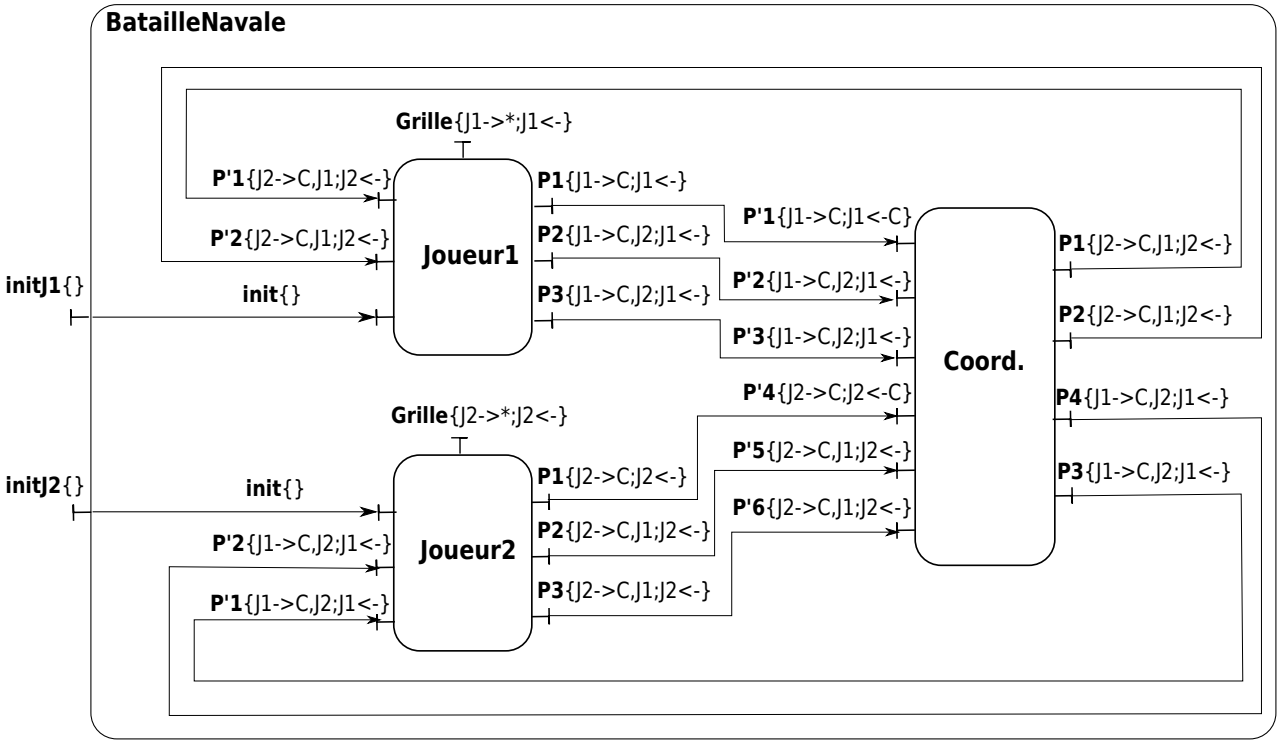


Figure IV.I.2 – Attribution des étiquettes de sécurité aux interfaces des composants du jeu de la bataille navale

Application de CIFIntra

Le code I.1 montre un extrait du fichier *Policy*.

Listing I.1 – Extrait du fichier *Policy* pour l'exemple de la bataille navale

```
<policy targetComposite="Battleship">
  <component name="Joueur1" instanceForIntraGen="true">
    <port name="P1" label="{J1->C ; J1<-C}" />
    <port name="init" label="{ }" />
    <attribute name="Grille" label="{J1->* ; J1<- }" />
    <capabilities><capability> C+ </capability></capabilities>
    ...
  </component>
  <component name="Coord">
    <port name="P'1" label="{J1->C ; J1<-C}" />
    ...
  </component>
   ...
</component>
```

Nous remarquons la présence de l'attribut *Grille* étiqueté $\{J_1 \rightarrow *; J_1 \leftarrow\}$. Cette grille doit être envoyée au coordinateur via le port P_1. Dans cet exemple, nous exécutons le générateur de code intra-composant sur le code du composant *Joueur*, qui est écrit dans une implémentation en Java de Fractal, appelée Julia [1]. Ce code sera étiqueté en utilisant les étiquettes de l'instance *Joueur1*, car l'attribut *instanceForIntraGen* est mis à *vrai*. En effet, dans les modèles orientés composants, nous pouvons créer plusieurs instances à partir d'une même définition de composant. Par exemple, nous définissons un seul composant *Joueur* à partir duquel nous générons deux instances *Joueur1* et *Joueur2*, puisque ces deux joueurs se comportent de la même manière et ont la même structure, ce qui veut dire que leur code d'implémentation est le même. Ainsi, la propagation des étiquettes dans le code peut être faite soit en utilisant les étiquettes de *Joueur1* soit celles de *Joueur2* (au choix). Grâce à l'attribut *instanceForIntraGen* de l'élément *component* dans le fichier *Policy*, nous avons choisi de propager les étiquettes relatives au *Joueur1*. Si cette même étiquette est mise à *vrai* pour le *Joueur2* également, le compilateur CIFIntra génèrera ainsi deux fichiers annotés, pour chacun des joueurs respectivement.

Le code I.2 représente un extrait du code généré par l'outil CIFIntra. Nous montrons dans ce qui suit comment est-ce que la propagation d'étiquette marche pour cet exemple.

1. **Julia** : Fractal Composition Framework Reference Implementation. http ://fractal.ow2.org/julia/

Listing I.2 – Code de la classe *Joueur* généré par CIFIntra

```
1  public class Joueur implements JoueurAttributes, JoueurInterface {
2    private Grille {J1->*;J1<-} grille;
3    private EnvoiGrilleItf {J1->C;J1<-} P1;
4
5    public void init(int{} nbBateaux) {
6      int{} numCouverts = 0;
7      for (int j=1; j< nbBateaux+1; j++) {
8        numCouverts += j;
9      }
10     final Bateau[{J1->*;J1<-}] {J1->*;J1<-} maStrategie = {
11           new Bateau(new Coordonnee(1,1),1,true),
12           new Bateau(new Coordonnee(1,3),2,false) };
13     int{J1->*;J1<-} i = 0;
14     for (int{} compt = numCouverts; compt>0 && grille!= null;) {
15        try{
16           Bateau {J1->*;J1<-} nouvPiece = maStrategie[i++];
17           if (nouvPiece!=null && nouvPiece.longueur>compt){
18             nouvPiece = new Bateau(nouvPiece.pos, compt,
19                     nouvPiece.estHorizontal);
20           }
21           grille.ajoutBateau(nouvPiece);
22           compt -= (nouvPiece == null ? 0 : nouvPiece.longueur);
23        }catch (ArrayIndexOutOfBoundsException ignore){
24        }catch (IllegalArgumentException ignore){ }
25     }
26     P1.envoiGrille(grille);
27   }
28 }
```

La première chose que le générateur de code fait est d'attribuer les étiquettes **immuables** aux ports et attributs du composant. Ceci est fait en parcourant le code du composant et en cherchant les attributs et méthodes dont les noms correspondent aux noms des interfaces et attributs définis dans l'instance CIForm. Cela nous permet dans cet exemple d'attribuer les étiquettes $\{J_1 \rightarrow *; J_1 \leftarrow\}$ à l'attribut *grille*, $\{J_1 \rightarrow C; J_1 \leftarrow\}$ à l'attribut P_1 et $\{\}$ au paramètre de la méthode *init*.

Par la suite, le compilateur parcourt le code pour vérifier les flux d'information entre les différentes variables et pour allouer des étiquettes de sécurité aux variables intermédiaires de manière adéquate. Il va donc tout d'abord chercher les instructions qui manipulent l'attribut *grille* : on peut retrouver une instruction à la ligne 21. Cette instruction permet d'ajouter un

nouveau bateau à la grille. La méthode *ajoutBateau* va ajouter l'objet *nouvPiece* à la liste de bateaux définis dans la classe Grille par l'attribut *bateaux*. Cet attribut n'étant pas immuable, puisque la classe Grille n'est pas un composant, son étiquette doit être calculée selon son utilisation dans le code. Ici, l'attribut *bateaux* de la classe Grille va être modifié à travers l'objet *grille*. Or, d'après le problème de référencement décrit dans la section I.1.2, partie III, les attributs d'un objet ne doivent pas être modifiés à partir de références plus restrictives. Ainsi, l'attribut *bateaux* doit avoir au moins l'étiquette $\{J_1 \rightarrow *; J_1 \leftarrow\}$. Puisque la variable *nouvPiece* lui est affectée, elle va donc avoir cette même étiquette.

D'un autre côté, à la ligne 26, la grille est envoyée via le port P_1, ce qui est illustré en Java par son passage en paramètre à une méthode de P_1. Ce passage en paramètre implique que l'objet P_1 est modifié par la variable *grille*, et donc selon la loi de non-interférence, cette variable ne doit pas être plus restrictive que lui. Or, si on compare les étiquettes de *grille* ($\{J_1 \rightarrow *; J_1 \leftarrow\}$) et de P_1 ($\{J_1 \rightarrow C; J_1 \leftarrow\}$), nous remarquons que :

$$\{J_1 \rightarrow *; J_1 \leftarrow\} \not\subseteq \{J_1 \rightarrow *; J_1 \leftarrow\}$$

Ainsi, la grille, qui a un niveau de confidentialité $\{J_1 \rightarrow *\}$ est envoyée via un port dont le niveau de confidentialité est moins restrictif ($\{J_1 \rightarrow C\}$), car il autorise au coordinateur de consulter le message reçu via P_1 alors que la grille est strictement confidentielle. En remarquant cette interférence, le compilateur fait appel au module Contrôleur, qui décide si cette interférence est autorisée ou pas. Le contrôleur consulte alors la liste de capacités du composant Joueur et remarque qu'il a la possibilité d'ajouter l'autorité C à ses étiquettes, grâce à la capacité C+. Il va donc autoriser la rétrogradation de la grille.

Le compilateur continue son traitement, jusqu'à ce que le code I.2 soit généré.

Application de CIFInter

Cette partie comporte d'abord la vérification du flux d'information inter-composants puis la génération de la description en ADL.

CIFInter vérifie en premier les interfaces reliées et compare leurs étiquettes. Par exemple, il compare les étiquettes du port $Joueur1.P_1$ et $Coord.P_1'$ pour voir si le message, qui circule de P_1 vers P_1', se dirige bien vers un port au moins aussi restrictif. Comme :

$$label(Joueur1.P_1) = \{J1 \rightarrow C; J1 \leftarrow\} \subseteq label(Coordinateur.P_1') = \{J1 \rightarrow C; J1 \leftarrow C\}$$

ainsi, la liaison est autorisée.

La deuxième étape est l'insertion des composants cryptographiques. La grille, envoyée par le joueur au coordinateur, doit être chiffrée avec la clef publique du coordinateur, et signée,

avant d'être envoyée. Dans ce cas, personne d'autre que le coordinateur ne sera capable de la consulter, et le coordinateur pourra s'assurer qu'elle n'a été modifiée par personne d'autre que le joueur.

Un composant cryptographique est alors inséré en amont de chaque composant fonctionnel, interceptant les données entrantes et sortantes. Dans notre exemple, toutes les interfaces nécessitent une opération de cryptographie, sauf l'interface *init*.

2 Réservation de billet d'avion par services web

Dans cet exemple, nous appliquons les outils CIF pour contrôler le secret des données dynamiquement créées dans une application de service web classique. Chaque donnée est équipée d'une étiquette spécifiant sa classification par rapport aux différentes catégories de sécurité.

Dans notre exemple, une telle étiquette englobera le degré de confidentialité de la destination. Considérons la situation suivante (inspirée de [Hutter06]) : un client utilisant les services web pour voyager en France. Vivant à Paris, il doit aller à Grenoble pour une mission secrète et veut garder son voyage confidentiel. En utilisant une interface graphique, il informe son service de réservation de ses intentions. Ce service web décompose le problème en service de voyage et en service de paiement. Le service de réservation contacte le service de voyage pour avoir la liste des différents vols disponibles et obtient en retour une information de vol spécifique ainsi que les prix correspondants. Ensuite, le service de réservation choisit un service de paiement qui réalisera le paiement du vol sélectionné.

La figure IV.I.3 illustre une représentation du système distribué. Suivant une architecture de composants de service SCA, chaque service est encapsulé dans un composant, dénommés respectivement R, T et P pour les services de réservation, de voyage et de paiement. Les composants sont reliés par deux sortes de liaisons selon le type de données échangées : une liaison permettant l'échange de données publiques, et une liaison *sécurisée* permettant l'échange de données secrètes. Le composant R, par exemple, a deux ports : *P1* avec une étiquette $\{R \rightarrow T, R \leftarrow T\}$ et *P2* avec une étiquette $\{R \rightarrow \perp, R \leftarrow T\}$ (avec $\perp$ représentant l'autorité la plus faible) qui peuvent être respectivement reliés aux ports *P'1* du composant T pour les données confidentielles, et à *P'2* du composant T pour les données publiques.

Dans notre scénario, tous les ports des composants sont configurés pour assurer l'intégrité des données, c'est à dire que deux propriétés doivent être assurées : celui qui envoie le message est celui qu'il prétend être, et un message ne peut pas être modifié par un attaquant. Pour faire face à ces menaces, la spécification *WS-Security*[2] applique une combinaison de signatures et de

2. **OASIS Web Site** : http ://www.oasis-open.org/

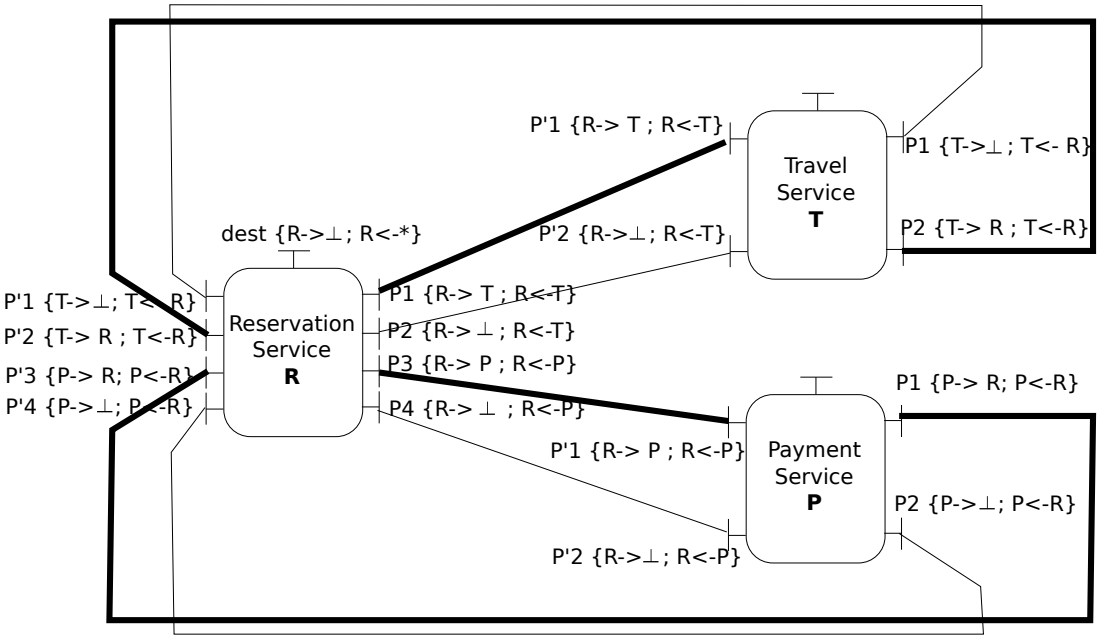


Figure IV.I.3 – Application de service web pour une réservation d'avion

jetons de sécurité pour démontrer et vérifier l'intégrité d'un message. Les signatures vérifient l'origine du message et les jetons de sécurité autorisent le traitement du message basé sur les créances associées au message. Les messages avec une signature invalide ou des jetons manquants sont rejetés. La spécification décrit la manière dont telles informations sont exprimées dans un format XML et dont elles sont incluses dans des enveloppes SOAP. Un ensemble de composants insérés permettent de signer ou chiffrer les messages. Le composant *sign* implémente l'authentification et les mécanismes d'autorisation en incluant les jetons correspondants dans l'entête de sécurité du message. Le choix entre de simples jetons (nom d'utilisateur/mot de passe en clair, nom d'utilisateur/empreinte numérique du mot de passe), des jetons binaires (certificats X.509, Kerberos) ou des jetons XML (Assertions SAML, XrML (*eXtensible Rights Markup Language*), XCBF (*XML Common Biometric Format*)) peut être configuré à travers les interfaces de contrôle et dépendent du contexte de l'application et de la qualité de service requise.

Puisque la destination de Grenoble est confidentielle, les données de réservation sont envoyées à travers le port *P1* du composant R : cela implique le chiffrement et la signature du message. La liste de vols, incluant leurs dates et prix renvoyés par le service de voyage, est envoyée sur le port *P1* du composant correspondant car elle est considérée comme étant confidentielle. Puisque le prix du vol est intuitivement une information publique, le développeur pourrait l'envoyer sur le port public *P2* du service de paiement. Toutefois, en utilisant les outils automatiques de CIF, il est facile de détecter que l'étiquette attribuée à la variable *prix* est

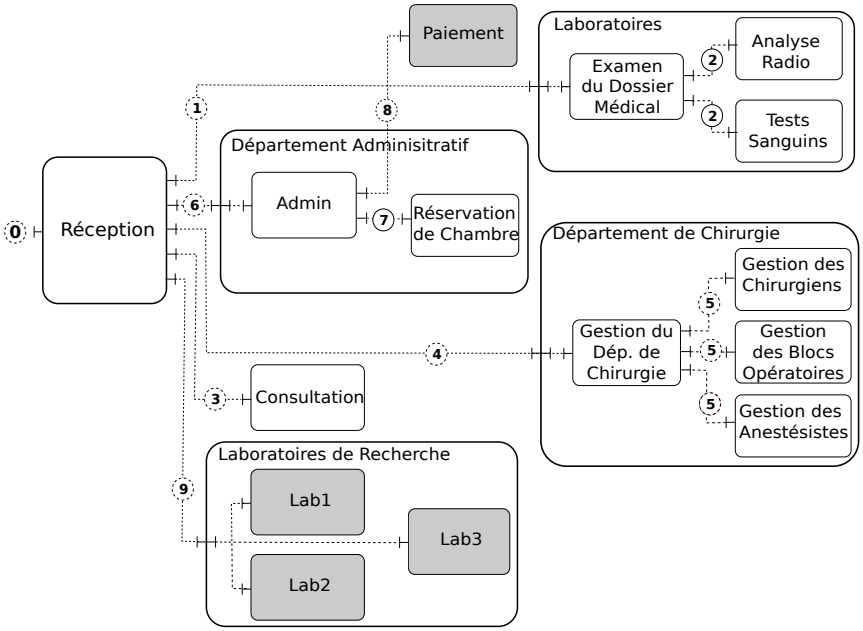


Figure IV.I.4 – Architecture de l'application de clinique de chirurgie esthétique

confidentielle, car cette donnée est calculée à partir d'une information confidentielle, qui est le type du vol. En effet, si un attaquant peut voir le prix du billet, il peut facilement deviner la destination du voyage, qui est confidentielle. Ainsi, la configuration sûre serait d'envoyer ce message sur un port confidentiel, soit *P1*, du service de paiement.

Cet exemple montre qu'en utilisant les outils CIF, la confidentialité d'une information renforce automatiquement la confidentialité des données calculées dynamiquement par les différents services web en planifiant le voyage.

3 Clinique de chirurgie esthétique

Description du système

Nous avons appliqué les outils CIF sur un système à base de composants existants. Ce système est réalisé en suivant la spécification SCA et représente les services offerts par une clinique de chirurgie esthétique. La figure IV.I.4 représente les différents composants de ce système.

La clinique contient un ensemble de départements internes, comme par exemple l'administration ou le département de chirurgie. Ces départements sont considérés comme des domaines de confiance, représentés à l'intérieur de composites, chacun contenant un ensemble de sous-composants représentant des entités qui réalisent une certaine activité.

Le composant principal qui communique avec le patient est le composant *Réception*. Il orchestre les communications entre l'ensemble des départements et retourne à la fin le résultat au patient.

Le processus de réservation suit les étapes suivantes : (0) Saisie des informations par le patient, (1) Envoi des informations aux laboratoires, (2) Analyse des tests sanguins et radios et envoi du résultat à la réception, (3) Étude du dossier par le praticien (4) Consultation du département de chirurgie, (5) Affectation de la salle d'opération, du chirurgien et de l'anesthésiste (6) Consultation de l'administration, (7) Réservation de la chambre pour le patient, (8) Calcul du montant et paiement et (9) Enregistrement de statistiques pour le laboratoire.

Les risques de sécurité

Supposons que les données qui circulent entre les composants peuvent être **personnelles** (nom, adresse, ...), **médicales** (identifiant du patient, date et type de l'opération...) ou **financières** (numéro de carte de crédit...). Chaque composant du système a accès à un certain type d'information. Le composant *Réception* a accès aux informations personnelles, médicales et financières, puisqu'il est le seul à communiquer avec le patient. Le composant *Administration* utilise également les données personnelles, médicales et financières, car il est impliqué dans les processus de paiement et de réservation de chambre. Le composant de *Paiement* utilise des données personnelles et financières. Le composant de *Réservation de chambre* a accès aux données personnelles et médicales, car il a besoin du nom du patient, ainsi que de la date de l'opération. Les *Laboratoires de recherche* ont besoin d'informations médicales pour les statistiques, mais aussi d'informations personnelles pour s'assurer que les données qu'ils reçoivent ne sont pas dupliquées. Enfin, le reste des composants (*Chirurgie*, *Consultation* et *Laboratoires d'analyse*) ont accès aux informations médicales uniquement.

En échangeant les informations, le système doit assurer que les contraintes précédentes soient respectées. Autrement dit, si un composant est habilité à consulter uniquement une catégorie de données, il ne doit pas être capable de déduire la valeur de celles appartenant à une autre catégorie. Cependant, dans le scénario détaillé ci-dessus, une *violation de confidentialité* peut être détectée.

Le composant *Paiement* reçoit de la part du composant *Administration* le montant à payer, en même temps que le numéro de carte de crédit, le nom et l'adresse du patient. Le problème réside dans le fait que le montant de l'opération est déduit des informations médicales du patient, ce qui permettrait au composant Paiement de deviner le type de l'opération esthétique que fait le patient.

Dans le scénario, nous détectons également une *violation d'intégrité*. Le composant *Laboratoire d'analyse radio*, par exemple, reçoit un dossier contenant l'historique médical du patient. Ce dossier est la propriété du patient et ne doit donc pas être modifié par qui que ce soit sauf lui. Toutefois, le composant *Laboratoire d'analyse radio* calcule les résultats de l'analyse en les déduisant du contenu de ce dossier. Dans ce cas, la clinique doit être explicitement autorisée à modifier des informations déduites à partir de données propres au patient.

Les mécanismes de sécurité classiques utilisés pour les systèmes distribués (contrôle d'accès, chiffrement, signature) ne peuvent pas résoudre à eux seuls ces types de problèmes, car on traite désormais de problèmes liés au contrôle de flux d'information, qui exige un suivi de l'information de bout en bout. En effet, si nous supposons que les procédures de contrôle d'accès nécessaires sont appliquées pour chaque composant et que les messages sont protégés durant le transport grâce à des fonctions cryptographiques adéquates, rien ne garantit que le comportement interne de chaque composant respecte les restrictions initiales du propriétaire de la donnée. Ce que nous avons vraiment besoin de faire est de garder la trace de l'information au delà du premier port qui la reçoit. Dans la section qui suit, nous montrons comment sécuriser l'application de la clinique esthétique avec CIF et expliquons la manière dont la violation de confidentialité décrites ci-dessus est détectée.

Sécuriser l'application avec CIF

Nous considérons un ensemble d'hypothèses de sécurité pour notre application :

- Quelques procédures de sécurité classiques sont assurées, comme par exemple l'authentification de l'utilisateur au niveau du composant *Réception*.
- Les composants patrimoniaux, comme le composant Paiement et les laboratoires de recherche, sont de confiance. Ils affichent leurs étiquettes de sécurité et mettent réellement en œuvre le niveau de sécurité qu'ils prétendent fournir.
- Nous considérons que le réseau dans un même domaine est sûr et qu'ainsi, aucune action cryptographique n'est nécessaire pour l'échange de données privées dans le même domaine.
- Nous supposons que les clés cryptographiques sont préalablement installées dans les machines des composants.

Modèle d'étiquettes

Nous utilisons pour cette étude de cas le modèle d'étiquettes à base de jetons décrit dans la section III.2.2, partie I.

Le niveau de confidentialité des données peut être déterminé par l'utilisation des jetons p (pour *personnel*), m (pour *médical*) et f (pour *financier*). Si une donnée est étiquetée m et

p, par exemple, cela veut dire qu'elle contient en même temps des informations médicales et personnelles. Dans ce cas, aucune information ne peut aller de cette donnée vers une autre donnée qui a le jeton m uniquement, car l'information personnelle dans la première va être divulguée à la seconde. Ce qui veut dire que $\{m\} \subseteq_C \{(m, p)\}$.

En ce qui concerne l'intégrité, nous considérons deux types de jetons : un jeton *pa* (pour *patient*), associé aux données qu'un patient peut modifier et un jeton *c* (pour *clinique*) associé aux données que tout composant appartenant à la clinique peut modifier. Supposons qu'une donnée a les jetons d'intégrité *pa* et *c*. Cela veut dire que cette donnée peut être modifiée soit par l'un soit par l'autre, mais également qu'elle est certifiée par les deux. Aucune information ne doit lui parvenir d'une donnée qui a uniquement le jeton *pa*, par exemple, car l'information qui était uniquement modifiable par le patient (comme par exemple le numéro de carte de crédit) devient modifiable par la clinique également, ce qui est considéré comme une atteinte à l'intégrité. Nous pouvons ainsi affirmer que $\{; (pa, c)\} \subseteq_I \{; pa\}$.

Configuration de sécurité

Le composant *Réception* contient un ensemble d'attributs représentant toutes les données initiales du patient. Il envoie ces données au composant *Administration*, via un port client étiqueté $\{(m, p, f); (pa; c)\}$. Le composant *Administration* envoie le montant de l'opération, le nom du patient et le numéro de la carte de crédit au composant *Paiement*, qui est un composant patrimonial. Ces données sont envoyées via un port étiqueté $\{(p, f); (pa, c)\}$, car nous avons confiance que le composant *Paiement* va préserver les données personnelles et financières du patient.

La figure IV.I.5-a montre un extrait du fichier ADL et du fichier Policy de l'application de la clinique, ainsi que du fichier ADL généré qui décrit les composants fonctionnels et les composants de cryptographie.

Application de CIFIntra

La figure IV.I.6 montre le fichier de débogage illustrant la propagation des étiquettes dans le code du composant *Administration*. Les étiquettes immuables sont représentées en caractères gras, alors que les étiquettes générées sont soulignées.

Le montant total de l'opération est une variable interne et sa valeur dépend de celle du type de l'opération, de la paie du chirurgien, du type de la chambre réservée (individuelle ou partagée) etc. Toutes ces informations sont étiquetées $\{(m, p, f); (pa, c)\}$, car elles ont été envoyées via un port portant cette étiquette. Comme ces données transmettent leur information à la variable

ADL SCA Initial

```
<composite name="aethetic-clinic" ...>
  <service name="Start" promote="ReceptionComponent/start"/>
  <component name="ReceptionComponent">
    <implementation.java class="clinic.reception.Reception"/>
    <service  name="Start">
        <interface.java interface="clinic.reception.StartService"/>
    </service>
    <reference name="consultationRef"
         target="ConsultationComponent/ConsultationService">
        <interface.java interface=
                "clinic.consultationDept.ConsultationService"/>
    </reference>
  </component>
  <component name="ConsultationComponent">
    <implementation.java class=
                "clinic.consultationDept.Consultation"/>
    <service  name="ConsultationService">
        <interface.java interface=
                "clinic.consultationDept.ConsultationService/>
    </service>
    .......
  </component>
</composite>
```

Fichier Policy

```
<policy targetComposite="aesthetic-clinic">
  <component name="ReceptionComponent">
    <port  name="Start" label="{ }" />
    <port  name="consultationRef" label="{(m);(pa,c)}" />
  </component>
  <component name="ConsultationComponent">
    <port  name="ConsultationService" label="{(m);(pa,c)}" />
  </component>
  ...
</policy>
```

a- Fichiers ADL et Policy Initiaux

ADL SCA Généré

```
<composite name="aethetic-clinic" ...>
  <service name="Start" promote="ReceptionComponent/start"/>
  <component name="ReceptionComponent">
    <implementation.java class="clinic.reception.Reception"/>
    <service  name="Start">
        <interface.java interface="clinic.reception.StartService"/>
    </service>
    <reference name="consultationRef"
         target="ReceptionComponent_security/ConsultationRef">
        <interface.java interface=
                "clinic.consultationDept.ConsultationService"/>
    </reference>
    .......
  </component>
  <component name="ReceptionComponent_security">
    <implementation.java class=
                "clinic.reception.ReceptionComponent_security"/>
    <service  name="ConsultationRef">
        <interface.java interface=
                "clinic.consultationDept.ConsultationService"/>
    </service>
    .......
    <reference name="send_consultationRef"  target=
        "ConsultationComponent_security/Receive_ConsultationService">
        <interface.java interface=
         "clinic.reception.SendItf_consultationRef_ConsultationService"/>
    </reference>
  </component>
</composite>
```

b- Fichier ADL SCA Généré

Figure IV.I.5 – Extrait de l'ADL CIF initial et de l'ADL SCA généré

```
package clinic.adminDept.impl;
...
public class Administration implements AdministrationService{
    private RoomReservationService{(m);(pa,c)} roomReservationRef;
    private PaymentService{(f,p);(pa,c)} paymentRef;
    ...
    public void sendPatientInformation ( CreditCard{(m,p,f);(pa,c)} cc,
                                         Operation{(m,p,f);(pa,c)} opType,
                                         SurgeonInfo{(m,p,f);(pa,c)} surgeon,
                                         RoomInfo{(m,p,f);(pa,c)} room,
                                         AnesthetistInfo{(m,p,f);(pa,c)} anes,
                                         BlocInfo{(m,p,f);(pa,c)} bloc){
        float amount{(m,p,f);(pa,c)} = surgeon.getWage() + opType.getCost()
                    + room.getPrice() + anes.getWage() + bloc.getPrice();
        paymentRef.performPayment ( cc, amount);
    }
    ...
}
```

*Interférence détectée ici, car la variable **amount** est envoyée via le port **paymentRef**, dont l'étiquette est moins restrictive*

Légende

Etiquettes de conf. :	**p**: personal
	f: financial
	m: medical
Etiquettes d'intégrité :	**pa**: patient
	c: clinic

Figure IV.I.6 – Extrait du fichier de débogage du composant Administration

amount, l'étiquette de cette dernière doit être au moins aussi restrictive que $\{(m, p, f); (pa, c)\}$. Ainsi, $label(amount) = \{(m, p, f); (pa, c)\}$.

Plus loin dans le code, ce montant doit être envoyé au composant *Paiement*, en même temps que le numéro de la carte de crédit, le nom et l'adresse du patient, via un port client portant l'étiquette $\{(p, f); (pa, c)\}$.

La réunion des étiquettes de toutes les informations à envoyer est :$\{(m, p, f); (pa, c)\}$, qui est plus restrictive que l'étiquette du port *paymentRef* via lequel elles sont acheminées, soit $\{(p, f); (pa, c)\}$. Cette interférence est détectée par l'outil CIFIntra. L'administrateur peut alors choisir d'autoriser l'interférence, ou de l'interdire et chercher une solution pour la corriger ou l'éviter. Dans notre cas, nous supposons que l'administrateur juge cette interférence dangereuse pour le patient et décide de la corriger en changeant l'étiquette du port *paymentRef* pour la rendre plus restrictive.

Application de CIFInter

L'étape suivante est d'appliquer l'outil CIFInter sur le système. En comparant les étiquettes des ports clients et serveurs associés, l'outil détecte une interférence entre le port *paymentRef* du composant *Administration* qui porte l'étiquette $\{(m, p, f); (pa, c)\}$, et le port serveur associé du composant *Paiement*, dont l'étiquette est restée à $\{(p, f); (pa, c)\}$. Cela implique qu'un flux d'information illégal a lieu : le composant *Paiement* garantit la préservation des informations personnelles et financières, mais pas des informations médicales. C'est pour cela que le concepteur doit modifier l'étiquette du port serveur du composant *Paiement*, puis *relancer l'outil CIFIntra* pour vérifier que les liaisons internes du composant Paiement, décrites dans son IBA, préservent le jeton m nouvellement ajouté à l'entrée du composant.

Une fois cette vérification réalisée, l'outil CIFInter est lancé une dernière fois pour que ses générateurs fonctionnels insèrent les composants de sécurité qui vont assurer le chiffrement et/ou la signature des messages sortants ainsi que le déchiffrement et/ou la vérification des messages entrants. La figure IV.I.5-b montre un extrait de la description SCA résultante. Les éléments surlignés en gris représentent les parties insérées dans le fichier ADL résultant par le compilateur CIFInter.

4 Calendrier partagé

L'application décrite dans cette partie permet à deux utilisateurs de fixer une date pour une réunion de manière quasi-automatique. Cette application à base de composants sera sécurisée en utilisant DCIF, avec le modèle d'étiquettes à base de jetons décrit dans la section III.2.2, partie

I. Nous supposons dans notre exemple que tous les composants sont dans le même domaine pour nous délivrer de toutes les opérations cryptographiques.

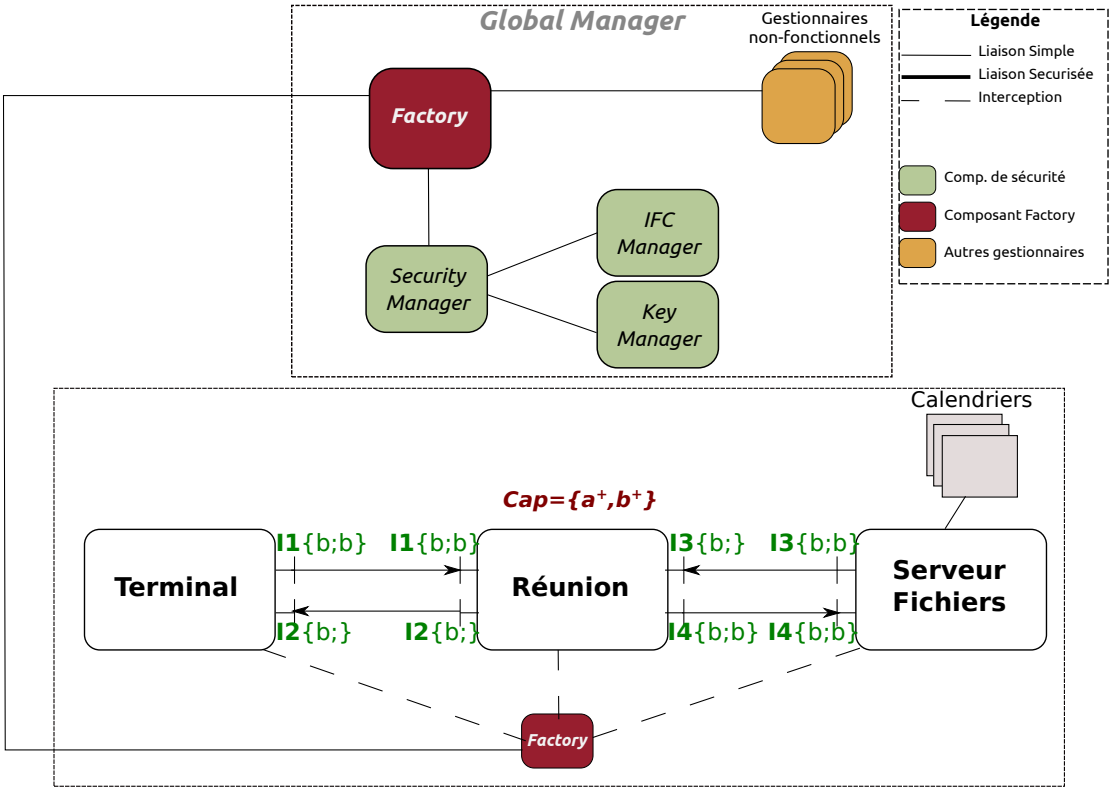


Figure IV.I.7 – Application de calendrier partagé sécurisée avec DCIF

La figure IV.I.7 présente l'architecture de l'application. Le composant *Terminal* fournit une interface aux utilisateurs pour qu'ils interagissent avec l'application. Le composant *Réunion* est en charge de la vérification de la disponibilité des utilisateurs à partir de leurs calendriers respectifs quand une nouvelle réunion est proposée. Enfin, le composant *ServeurFichiers* contient le code qui accède aux calendriers des utilisateurs. Le terminal peut indiquer au composant *Réunion* les requêtes des utilisateurs via son interface cliente *I1*, et recevoir sa réponse via l'interface serveur *I2*. Le composant *Réunion*, d'autre part, consulte le composant *ServeurFichiers* pour avoir des information sur les calendriers des utilisateurs en utilisant son interface *I4*, et le serveur de fichiers lui renvoie la réponse via l'interface *I3*[3].

Considérons le scénario suivant : le utilisateurs Alice et Bob gardent leurs calendriers privés respectifs dans leurs fichiers privés *Cal_Alice* et *Cal_Bob*. Bob veut programmer une réunion avec Alice, il suggère ainsi quelques dates et attend une réponse de sa part. Le composant

3. Par souci de lisibilité, les interfaces communicantes sont placées l'une en face de l'autre sur la figure, au lieu de placer les interfaces serveur à gauche et les interfaces clientes à droite des composants. Les sens de communication sont indiqués par des flèches.

Réunion examine ces suggestions, en considérant les informations lues à partir des fichiers *Cal_Alice* et *Cal_Bob*, puis informe le terminal si ces réunions sont possibles ou pas. Si Bob est un utilisateur malicieux, il peut déduire des informations sur le calendrier d'Alice en testant plusieurs propositions de réunions ce qui peut conduire à un problème d'interférence. Le challenge ici est de permettre au composant *Réunion* de combiner les données privées d'Alice et Bob sans divulguer des informations, à moins qu'Alice ne choisisse de révéler ses données privées à Bob, si elle le considère comme utilisateur de confiance.

Au déploiement ou à la reconfiguration, le Factory consulte le IFCM pour vérifier la validité des composants et liaisons. Pour cela, nous définissons les jetons a et b, représentant respectivement les données secrètes d'Alice et de Bob. Si un niveau de confidentialité contient le jeton a, par exemple, cela veut dire qu'il contient des informations secrètes d'Alice. Si un niveau d'intégrité contient l'étiquette a, cela veut dire que cette information est teintée par l'utilisateur Alice.

Dans le scénario précédent, la configuration initiale des étiquettes est mise de sorte que l'utilisateur connecté (Bob, par exemple) puisse consulter son propre calendrier, ce qui veut dire que la liaison entre l'interface *Terminal.I2* (portant l'étiquette $\{b;\}$) et l'interface *Réunion.I2* (portant la même étiquette) est établie, puisqu'aucun risque de révéler les données privées n'est présent. Ainsi, l'utilisateur peut avoir une réponse pour chacune de ses requêtes et exploiter ainsi le contenu de son calendrier. De plus, les capacités du composant *Réunion* ont définies pour $Cap = \{a^+, b^+\}$, car nous considérons que ce composant peut autoriser l'ajout d'une teinte supplémentaire à une information, mais interdit d'enlever une teinte existante. On suppose également ici que la liste de confiance est vide, c'est à dire que, pour commencer, aucun utilisateur ne fait confiance à un autre. De plus, initialement, chaque calendrier de l'utilisateur x porte l'étiquette $\{x; x\}$.

Supposons maintenant que Bob veut proposer une réunion à Alice. Le composant *ServeurFichiers* va consulter le calendrier d'Alice. Par conséquent, les interfaces de ce composant doivent avoir des étiquettes au moins aussi restrictives que celles du calendrier d'Alice. Ainsi :

$$Label(ServeurFichiers.I3) = Label(ServeurFichiers.I4) = \{a; a\}$$

Puisque l'interface *ServeurFichier.I3* est liée à *Réunion.I3*, sa modification entraîne la vérification de *Réunion.I3*, pour voir si la liaison reste valide ou pas. Puisque :

$$Label(ServeurFichiers.I3 = \{a; a\})\ et\ Label(Réunion.I3 = \{b;\})$$

alors une violation de la confidentialité est détectée, car $\{a; a\} \not\subseteq_C \{b;\}$.

Suite à la détection de cette violation, le IFCM doit consulter les capacités du composant *Réunion* pour vérifier si une rétrogradation de l'interface *Réunion.I3* est possible. Cela revient à vérifier si le jeton a peut être ajouté au niveau de confidentialité de cette interface, de manière à ce qu'elle accepte les données privées d'Alice. Nous remarquons que le composant *Réunion* a la capacité a^+, il peut donc rétrograder le niveau de son interface *I3* vers $\{a, b; \}$.

Ceci étant fait, aucune violation de confidentialité ou d'intégrité n'est détectée, et la liaison, devenue valide, est établie. Autoriser cette liaison veut dire que le composant *Réunion* est autorisé à manipuler et transférer des données privées d'Alice.

Puisque chaque changement d'étiquettes entraîne une re-vérification des composants, le composant *Intra-Component Verifier* du IFCM va propager les nouvelles étiquettes dans le code du composant *Réunion*, ce qui induit le changement des étiquettes des interfaces de sortie (*I2* et *I4*) du composant *Réunion*, car les informations acheminées par ces interfaces dépendent directement des valeurs lues par l'interface d'entrée *I3*. D'où, nous obtenons les configurations suivantes :

$$Label(Réunion.I2) = Label(Réunion.I4) = \{a, b; \}$$

Selon les règles de la non-interférence, la liaison entre *Réunion.I2* et *Terminal.I2* devient invalide car $\{a, b; \} \not\subseteq_C \{b; \}$. De même pour la liaison entre *Réunion.I4* qui porte l'étiquette $\{a, b; \}$ avec *ServeurFichiers.I4* qui porte l'étiquette $\{a; a\}$, qui est invalide car $\{a, b; \} \not\subseteq \{a; a\}$.

Le IFCM consulte ensuite les capacités du composant *Réunion* et vérifie s'il a la possibilité de rétrograder les données privées de Bob (pour les écrire dans le fichier privé d'Alice) ou rétrograder celles d'Alice (pour les afficher à Bob dans le Terminal). Or, cela n'est pas possible, car le composant *Réunion* n'a pas les capacités a^- ni b^-. De plus, la liste de confiance étant vide, rien ne permet de dire si l'utilisateur Alice fait confiance à l'utilisateur Bob pour l'autoriser à consulter ses données, et vice versa. Par conséquent, la violation de confidentialité persiste, et ces liaisons sont mises *en attente*. L'application protège ainsi les données privées d'Alice, et empêche Bob d'avoir une idée sur la disponibilité d'Alice pour la date qu'il a proposé. Il doit donc attendre qu'Alice se connecte pour qu'elle lui envoie sa réponse.

Conclusion

Dans cette partie, nous avons illustré l'utilisation des outils CIF et DCIF par divers exemples, qui diffèrent par leurs types, leurs modèles orientés composants et leurs modèles d'étiquettes. Ces exemples ont servi à montrer l'utilité et la faisabilité de notre approche et son application sur des cas concrets. Cependant, aussi utile soit-il, tout mécanisme de sécurité entraîne

en général un coût supplémentaire en termes de performances. La partie suivante nous permet d'évaluer le surcoût entraîné par la vérification de la non-interférence avec l'outil CIF.

D'autre part, il est important de montrer que les outils que nous avons réalisé permettent de garantir réellement la non-interférence pour l'ensemble du système distribué. Nous montrons deux techniques de vérification formelles que nous appliquons sur nos outils pour montrer que CIF produit bien un système non-interférent.

Chapitre II

Évaluation des performances

Dans ce chapitre, nous évaluons les outils CIF en termes de taille de code, de coût de compilation, de coût de configuration et de surcout à l'exécution. L'évaluation est réalisée sur un Macintosh Intel Core 2 Duo 2GHz, avec 4GB de RAM, utilisant MacOSX 10.5.8. Tous les exemples sont conçus avec le standard SCA en utilisant l'implémentation Frascati. L'évaluation a été réalisée avec le profileur de code YourKit [1] pour l'estimation des temps de compilation et le plugin Eclipse Software Metrics [2] pour l'estimation de la taille du code. Pour chaque étape d'évaluation, nous vérifions les résultats obtenus sur l'application de la clinique de chirurgie esthétique, car elle peut nous aider à avoir une idée sur le coût des outils CIF sur une application réaliste.

1 Implémentation et taille de code

Les outils CIFIntra et CIFInter sont réalisés en Java 1.6. Pour CIFIntra, nous étendons le compilateur Polyglot pour l'analyse et la génération du code. Pour CIFInter, nous utilisons l'analyseur DOM fourni dans l'API *javax.xml*, pour parcourir et générer des documents XML. La taille totale du code de CIF est de 4917 LoC (Lines of Code). Le générateur CIForm comprend 928 LoC, l'outil CIFInter 1422 LoC dont le tiers est dédié à la classe du générateur et CIFIntra 2567 LoC, partitionnés assez équitablement entre l'ensemble des classes visiteurs.

1. **YourKit Java Profiler** : http ://www.yourkit.com/
2. **Software Metrics Eclipse Plugin** : http ://metrics.sourceforge.net

2 Coût de la configuration

La configuration consiste à attribuer des étiquettes aux différentes parties du système (attributs et ports). Nous avons repris le code d'une application de bataille navale [Myers06] développée avec JIF. Nous avons réécrit le système sous forme de composants. Nous obtenons ainsi trois composants : le coordinateur du jeu et deux joueurs. Chaque communication entre ces trois partis se fait grâce à des liaisons et à travers des ports. Nous avons utilisé le modèle Fractal pour représenter le système. La configuration de la sécurité a nécessité l'attribution de 11 étiquettes appliquées au niveau des ports et attributs dans le fichier Policy, contre 143 étiquettes réparties sur toute l'implémentation dans le code initial écrit en JIF, sans compter les instructions de rétrogradation ou d'attribution d'autorité [Myers00]. Avec CIF, il n'est pas nécessaire d'annoter toutes les variables, ce qui simplifie grandement la configuration de la sécurité. De plus, cette configuration a l'avantage d'être complètement séparée du code et de l'architecture de l'application.

3 Coût de la compilation

3.1 Coût de la compilation de CIFIntra

La complexité de notre outil de génération intra-composant peut être évaluée par le nombre de passages sur le code d'implémentation. Pour propager les étiquettes dans le code du composant, ce dernier est scanné autant de fois que les étiquettes générées changent. En effet, le compilateur s'arrête quand il détecte une interférence, ou bien quand il atteint un état stable où les étiquettes ne changent plus.

Chaque fois que l'étiquette d'une variable change, l'outil revérifie que ce changement n'affecte pas le reste du code. Le nombre de passages sur le code augmente considérablement avec la taille du code, bien sûr, mais surtout avec le nombre et l'utilisation des variables intermédiaires. Cela est dû au fait que l'un des rôles de CIFIntra est d'assigner des étiquettes à ces variables. Le code doit donc être scanné autant de fois que nécessaire pour étiqueter convenablement la variable. Prenons par exemple la portion de code suivante (a étant une variable locale, b et c sont des attributs) :

$$while(true)\{b = a; a = c;\}$$

Supposons que $label(c) \not\subseteq label(b)$. Le premier passage du compilateur trouvera que a n'a pas d'étiquette et lui assignera celle de c (car si $a = c$, alors $label(a)$ doit être au moins aussi restrictif que $label(c)$) et continuera sans lancer d'exception, même s'il est clair pour le lecteur

que c'est un cas d'interférence, car à la seconde itération de la boucle, b prendra la valeur de c, ce qui est interdit. C'est pourquoi le compilateur doit faire deux passages sur cette portion : dans la seconde, il détectera que l'affectation $b = a$ est illégale, car un flux d'information est transmis de a vers b alors que $label(a) \not\subseteq label(b)$.

Le nombre de passages dépend également du nombre de blocs imbriqués. En effet, les blocs doivent être scannés récursivement pour vérifier que le compteur programme devient uniquement de plus en plus restrictif, évitant ainsi les flux implicites.

Nous pouvons déduire à première vue que cet algorithme termine, car (1) le nombre d'étiquettes à modifier est fini, (2) les valeurs des étiquettes changent de manière incrémentale (du moins vers le plus restrictif) et (3) les étiquettes sont plafonnées par une borne supérieure, qui est celle de l'étiquette la plus restrictive du treillis selon le modèle défini par l'utilisateur.

3.2 Temps d'exécution de CIFInter

Nous avons développé deux benchmarks (représentés dans la figure IV.II.1) pour un test de montée en charge :

- **Bench1** : Une simple application formée d'un nombre variable de composants minimalistes connectés. Différentes dispositions sont testées et, puisque l'architecture du système n'a aucun impact sur le temps d'exécution de CIFInter, nous présentons les résultats avec des composants connectés en série, échangeant un message de taille 10K. Une simple liaison locale relie deux composants successifs et induit à chaque fois l'insertion de composants cryptographiques (pour le chiffrement et la signature).
- **Bench2** : Une application composée de deux composants, un client et un serveur, connectés avec un nombre variable de liaisons.

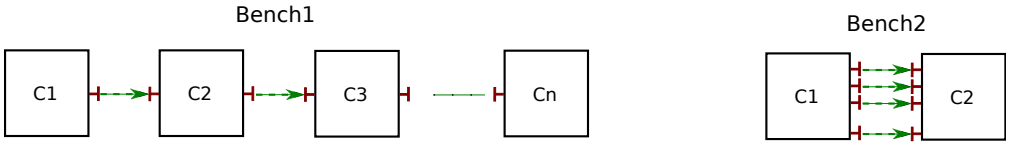


Figure IV.II.1 – Illustration des deux benchmarks utilisés pour les tests de montée de charge

Nous mesurons avec ces deux benchmarks le temps nécessaire pour la vérification du fichier Policy, pour l'encapsulation des composants cryptographiques et la génération du nouveau fichier ADL fonctionnel. Pour faire varier le nombre de composants et de liaisons de manière significative, nous avons réalisé un programme qui permet de générer automatiquement les fichiers ADL de ces benchmarks, selon un nombre de composants variable pour le Bench1 et de liaisons variable pour le Bench2.

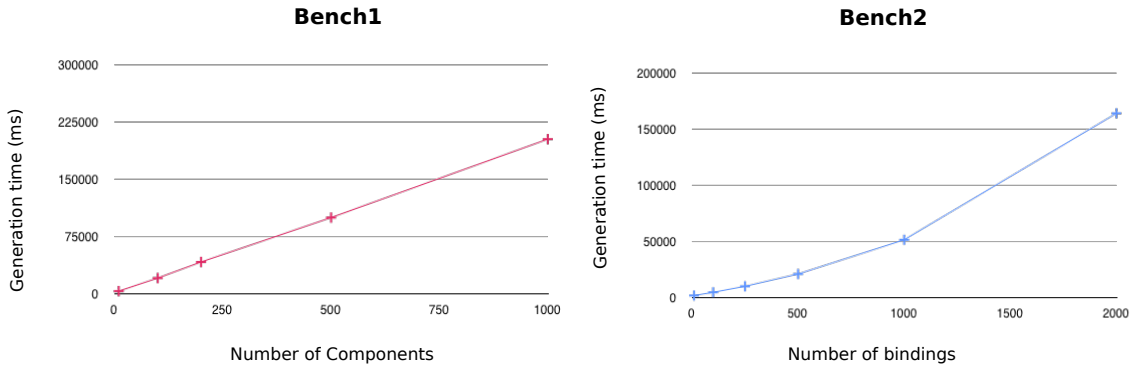


Figure IV.II.2 – Courbes représentant la montée en charge pour les deux benchmarks utilisés

Quand le nombre de composants augmente, les temps de vérification et de génération de CIFInter augmentent de manière linéaire, avec un temps de compilation qui reste acceptable (jusqu'à 220 secondes pour 1000 composants). Le temps de compilation varie également de manière proportionnelle au nombre de liaisons. En effet, un composant cryptographique commun pour toutes les liaisons est généré et pour chaque liaison, une interface appropriée est ajoutée (pour 500 liaisons, nous mesurons 20 secondes pour l'exécution de CIFInter).

3.3 Coût de la compilation pour l'application de la clinique

Le temps de compilation nécessaire pour réaliser une vérification intra-composant pour le composant Réception (150 LoC) est évalué à 1992 ms. Mais même si ce temps de compilation varie selon la taille de la classe et le nombre de variables, il reste acceptable, car la génération est réalisée une seule fois au déploiement.

En ce qui concerne CIFInter, le temps de compilation pour la vérification des liaisons et la génération du composant de cryptographie est estimé à 4838 ms, avec à peu près 75% du temps pour la génération de l'implémentation des composants cryptographiques. Comme un seul composant cryptographique est associé à chaque composant fonctionnel, l'implémentation dépend du nombre de ports clients/serveurs de ce composant fonctionnel, qui envoie/reçoit des messages nécessitant un chiffrement ou une signature.

4 Surcoût sur le temps d'exécution

Le choix de la méthode cryptographique utilisée pour la signature et le chiffrement des messages a clairement un impact sur la performance de l'application. Dans notre prototype, tous les messages sont considérés comme des objets Java, chiffrés avec la classe *SealedObject* de l'API *security* de Java, avec l'algorithme asymétrique RSA et une taille de clé de 2048 bits. Pour

la signature, l'algorithme utilisé est MD5/RSA, avec la classe *SignedObject*. La performance du système résultant peut être améliorée si le concepteur choisit d'utiliser, par exemple, un cryptage symétrique au lieu de l'asymétrique actuellement utilisé ; elle peut être améliorée encore plus si des composants matériels sont utilisés pour la génération et le partage de clefs. Cette optimisation est orthogonale à notre travail sur le framework CIF.

Dans le cas de l'application de la clinique, l'ajout de 6 composants cryptographiques induit un surcoût de 20% : pour le scénario décrit dans le chapitre section I.3, partie IV, le temps d'exécution augmente de 6400 ms sans cryptographie à 7675 ms avec cryptographie.

Dans CIF, la séparation de la sécurité du code fonctionnel permet l'optimisation et la modification des protocoles cryptographiques sans toucher au code fonctionnel, mais peut induire un surcoût sur le temps d'exécution. Nous avons mesuré ce surcoût et avons trouvé qu'il était négligeable comparé au coût de la cryptographie. En effet, comme les composants cryptographiques insérés sont des composants locaux, le surcoût induit est équivalent à celui d'un appel de méthode.

Le tableau II.1 récapitule l'ensemble des valeurs que nous avons relevées.

Tableau II.1 – Évaluation des performances

Application				Tools			
	Before CIF	After CIF	Scale		CIForm	CIFIntra	CIFInter
LoC	1068	4475	x 4,19	LoC	928	2567	1422
Components	13	26	x 2	HotSpots	Label	Visitors	Comp. gen.
Exec Time	6400ms	11175ms	x 1,74	Exec Time	1164ms	1992 ms	4838 ms
Memory	35 MB	43 MB	x 1,22	Memory	23 MB	14,2 MB	28 MB

Conclusion

L'étude de performances réalisée montre que le surcoût de la vérification de la non-interférence à la compilation n'est pas très important, et que c'est la cryptographie qui pénalise le plus les performances du système.

Un autre aspect de l'évaluation doit être abordé, c'est la vérification formelle de la non-interférence. Nous présentons dans le chapitre suivant des techniques de vérification que nous appliquons sur notre canevas CIF.

Chapitre III

Étude Formelle

Nous proposons dans ce chapitre une méthode pour montrer que la propagation des étiquettes dans le code d'un composant, l'une des contribution les plus importantes dans notre solution, produit bien un composant non-interférent. Pour cela, nous utilisons les systèmes de réécriture. Nous appliquons la technique utilisée par [Alba-Castro09], pour montrer comment est-ce qu'on peut prouver que le code généré par CIFIntra est bien non-interférent.

1 Logique de Réécriture

La logique de réécriture [Klop92, Dershowitz89, Meseguer92] présente un ensemble de méthodes pour la manipulation et la transformations des expressions, valeurs, propriétés et méthodes. Elle se base sur le remplacement d'un ensemble de termes par un autre dans une formule. Les systèmes de réécriture (TRS pour *Term Rewriting System*) offrent une syntaxe et une sémantique simples pour faciliter l'analyse mathématique pour la spécification des protocoles de sécurité, la résolution de contraintes, les systèmes de transition, la transformation de programmes, etc.

Une **signature** dans la logique de réécriture est une théorie équationnelle (Σ, E) où Σ est une signature équationnelle et E un ensemble de Σ-équations.

La réécriture permet de construire des classes d'équivalence de **termes** modulo E. Un terme $t \in \mathcal{T}$ est un élément d'une équation pouvant être **clos** (contenant uniquement des symboles fonctionnels) ou **ouvert** (contenant des symboles fonctionnels et des variables).

Soit une signature (Σ, E), les phrases de la logique de réécriture sont des séquences de la forme : $[t]_E \longrightarrow [t']_E$, où t et t' sont des Σ-termes pouvant contenir des variables et $[t]_E$ montre la classe d'équivalence de t modulo les équations E.

La théorie de réécriture comprend 4 tuples : (Σ, E, L, R) :
– (Σ, E) : La théorie équationnelle modulo laquelle on réécrit.
– L : Un ensemble d'étiquettes (*Labels*).
– R : Un ensemble de règles étiquetées pouvant être conditionnelles (*Rules*).
On peut définir une règle de réécriture comme suit :

Définition 4 (Règle de réécriture)**.** Pour deux termes l et r de l'ensemble de termes $\mathcal{T}$ tel que $Var(l) \subseteq Var(r)$, $l \longrightarrow r$ est une règle de réécriture.

Exemple. La règle de réécriture $g(x) \longrightarrow f(a, f(b, x))$ avec x une variable, nous permet de réécrire le terme $g(a + b)$ en $f(a, f(b, a + b))$.

Définition 5 (Pas de réécriture)**.** Pour un système de réécriture donné R, on note $t \longrightarrow_R t'$ un **pas de réécriture** avec une des règles de R.
On note $t \xrightarrow{l \to r} t'$ si c'est la règle de réécriture $l \longrightarrow r$ qui est utilisée.

Définition 6 (Atteignabilité)**.** $t \longrightarrow *_R t'$ décrit le fait que t' est **atteignable** par réécriture à partir de t avec un nombre fini d'étapes.

Exemple. Soit le système de réécriture suivant :
$R = \{f(x, g(d)) \longrightarrow f(c, g(d)), g(x) \longrightarrow x, c \longrightarrow d, f(x, x) \longrightarrow x\}$.
En appliquant la logique de réécriture sur le terme $f(b, g(c))$, on obtient :

$$\begin{aligned} f(b, g(c)) &\xrightarrow{c \to d} f(b, g(d)) \\ &\xrightarrow{f(x,g(d)) \to f(c,g(d))} f(c, g(cd)) \\ &\xrightarrow{c \to d} f(d, g(d)) \\ &\xrightarrow{g(x) \to x} f(d, d) \\ &\xrightarrow{f(x,x) \to x} d \end{aligned}$$

On en déduit ainsi que d **est atteignable à partir de** $f(x, g(d))$ et on note :

$$f(x, g(d)) \longrightarrow *_R d.$$

2 Réécriture et protocoles de sécurité

Les systèmes de réécriture peuvent être utilisés pour vérifier certaines propriétés de sécurité, telles que la confidentialité[Denker98], l'authentification [Ogata04], la non-répudiation [Li07] et plus récemment la non-interférence [Alba-Castro09, Alba-Castro10].

Un problème de sécurité peut, en général, se réduire à un problème d'atteignabilité en réécriture, qui est un problème **indécidable**, c'est à dire qu'on ne peut ni le prouver, ni prouver sa négation.

Des méthodes par approximations peuvent alors être utilisées, ce qui peut transformer l'algorithme de vérification en semi-algorithme. On peut ainsi définir l'ensemble des configurations interdites, représentant les états du système qui présentent un risque de sécurité, et déterminer si oui ou non ces configurations sont atteignables.

Prenons par exemple l'ensemble des configurations accessibles $\mathcal{R}^*(D)$, où $\mathcal{R}^*$ est la clôture transitive et réflexive de $\mathcal{R}$ et D un ensemble de configurations de départ. Soit $\mathcal{A}$ un ensemble de configurations interdites. Si on a $\mathcal{R}^*(D) \cap \mathcal{A} = \emptyset$, alors aucune configuration de $\mathcal{A}$ n'est atteignable, et donc la propriété de sécurité est vérifiée. Cependant, en ce qui concerne la vérification des protocoles de sécurité, le langage $\mathcal{R}^*(D)$ ne termine pas, et l'ensemble des termes initiaux D peut être infini [Genet09].

Cependant, il est possible d'appliquer des restrictions sur le langage $\mathcal{R}$ pour un ensemble de termes initiaux défini, pour aboutir à un ensemble de termes atteignables. Cela nous amène à proposer des approximations de $\mathcal{R}^*(D)$ pour semi-décider si la propriété est vérifiée ou pas.

Nous appliquons pour notre travail un système de réécriture défini par [Alba-Castro09] pour vérifier la propriété de non-interférence sur les systèmes écrits en Java. Nous utilisons pour cela le langage Maude.

3 Maude

Maude [1] [Clavel07] est un langage de réécriture développé par SRI International. Il supporte à la fois la logique équationnelle et la logique de réécriture pour un grand nombre d'applications.

Une spécification Maude a deux parties :

- Une partie composée d'**équations** décrivant la structure et les propriétés des états d'un système, en utilisant des **modules fonctionnels** représentés comme suit :

  fmod *nom_module* **is** ... **endfm**

- Une partie composée de **règles** spécifiant la manière dont le système peut changer dans le temps, et cela en utilisant des **modules systèmes** représentés comme suit :

  mod *nom_module* **is** ... **endm**

1. **Maude** : http ://maude.cs.uiuc.edu/

3.1 Modules fonctionnels

Les types de données abstraits (ADT pour *Abstract Data Types*) sont spécifiées dans Maude grâce aux modules fonctionnels. Ils représentent les différents éléments ainsi que les opérations sur ces éléments. On en distingue quatre types : les **importations** (*imports*), les **types** (*sorts*), les **déclarations d'opérateurs** (*opdecls*) et les **équations** (*eqns*). Ils sont ordonnés comme suit dans les modules fonctionnels.

fmod *nom_module* **is**
    <imports> ***réutilisation, modularité*
    <sorts> ***types et sous-types de données*
    <opdecls> ***noms et arités des opérations*
    <eqns> ***comment calculer les fonctions*
endfm

Types (*sorts*) Grâce à l'ADT *sort*, on peut déterminer les types de données et les ordonner par sous-types (*subsorts*).

fmod Animaux **is**
    ...
    sort Animal .
    subsort Dog < Animal .
    sorts Terrier Hound .
    subsorts Terrier Hound < Dog .
    ...
endfm

Déclaration d'opérateurs (*opdecl*) Les opérations peuvent être définies entre les types grâce au mot clef "*op*". Une déclaration d'opérateur a la forme suivante :

op *nom_opérateur* **:** *types_arguments* **->** *type_résultat* [*attributs*] .

– Le nom de l'opérateur peut être soit sous forme de **préfixe**, tel que l'opérateur d'addition *+(x,y)* :

    op + **:** Nat Nat **->** Nat .

  Nat étant le type *entiers naturels* ; ou alors sous la forme **mixfix**, tel que l'opérateur d'addition *x+y* :

    op _+_ **:** Nat Nat **->** Nat .

- Les attributs sont utilisés pour désigner des fonctions particulières telle que :
  * [**ctor**] : désigne un constructeur
  * [**assoc**] : désigne un opérateur associatif
  * [**comm**] : désigne un opérateur commutatif
  * [**id :***terme*] : désigne le terme *nul* de l'opérateur (résultant en une identité).
- Les opérateurs peuvent être surchargés :

  op _+_ **:** Nat Nat **->** Nat .
  op _+_ **:** Integer Integer **->** Integer .

Équations (*eqns*) Les équations fournissent l'interpréteur de Maude avec un certain nombre de règles pour simplifier une expression. La syntaxe utilise le mot clef "*eq*", suivi par deux expressions séparées par le symbole "=". Voici un exemple :

fmod PEANO-NAT-ADD **is**
  sort Nat .
  op 0 **:** Nat [ctor] .
  op s **:** Nat **->** Nat [ctor] .
  op _+_ **:** Nat Nat -> Nat .
  eq s(M) + N = s(M+N) .
endfm

Commande de réduction (*red*) La commande de réduction *red* calcule la valeur d'un terme en utilisant les équations de la gauche vers la droite jusqu'à ce qu'aucune équation ne puisse être appliquée.

Maude > load PEANO-NAT-ADD.maude
Maude > red s(0) + s(s(0)) .
Maude > **result** Nat : s(s(s(0)))

Pour voir toutes les étapes de la réduction, on tape la commande :

Maude > set trace on .

Importation de modules (*import*) Un module M peut importer d'autres modules (sous-modules) en utilisant trois modes différents :
- *protecting* : utiliser seulement sans modifier.
- *extending* : étendre uniquement avec des constructeurs.

– *including* : modifier les comportements des ADT et des règles de déduction.

```
fmod PEANO-NAT-MULT is
        protecting PEANO-NAT-ADD .
        op _*_ : Nat Nat -> Nat .
        vars M N : Nat .
        eq N * 0 = 0 .
        eq N * s(M) = N + (N * M) .
endfm
```

Équations conditionnelles (*ceq*) Les équations conditionnelles exécutent une réduction si et seulement si la condition se réduit à *vrai*.

```
ceq N - M = 0 if M > N .
eq max (M,N) = if N > M then N else M fi .
```

3.2 Modules système

La partie dynamique d'un système est spécifiée grâce aux *règles de réécriture*. Elles permettent de définir un ensemble de règles de calcul sur les ADT spécifiés par la partie fonctionnelle.

```
mod nom_module  is
        **partie fonctionnelle
        <imports>    **réutilisation, modularité
        <sorts>      **types et sous-types de données
        <opdecls>    **noms et arités des opérations
        <eqns>       **comment calculer les fonctions

        **partie dynamique
        <rules>      **règles de réécriture
endm
```

Règles de réécriture *rl* Une règle de réécriture déclare la relation entre les états et les transitions. Elles sont irréversibles. Ces règles sont définies par l'une de ces formes :

```
rl[id] : term_gauche => term_droite .
crl[id] : term_gauche => term_droite if cond .
```

Une règle s'applique à un terme T s'il existe une substitution S (association de variables à des termes) tel que S(*terme_gauche*) est un sous-terme de T et S(*cond*) se réécrit *vrai*. Dans ce cas, T peut être réécrit en remplaçant le sous-terme adéquat par l'instance de *terme_droite* correspondante.

Commande de réécriture *rew* La commande de réécriture peut simuler au plus n pas d'un comportement possible à partir d'un état initial t :

Maude > rew [n] t .

Le nombre de pas maximum [n] peut être omis si le système termine. Voici un exemple illustrant l'utilisation de la réécriture.

```
mod COMPTER-CIGARETTES is
        protecting Nat .
        sort Etat .
        op c : Nat -> Etat [ctor] .
        op b : Nat -> Etat [ctor] .
        op _ _ : Etat Etat -> Etat [ctor assoc comm] .
        vars W X Y Z : Nat .
        rl[fumer] c (X) => b(X + 1) .
        rl[nouvelle] b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .
endm

** Commande de réécriture
Maude > rew [100] c(0) c(0) c(0) c(0) c(0) c(0) c(0)
                  c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) .

** Résultat
Maude > b(21)
```

4 Vérification de la non-interférence avec un système de réécriture

[Alba-Castro09] définit une extension de la sémantique de Java avec les flux d'information en utilisant le langage Maude. Il utilise une version abstraite et à états finis de la sémantique

opérationnelle du flux d'information pour vérifier le programme. Il produit à la suite de cette vérification un certificat de non-interférence, qui est un ensemble de preuves de réécriture qui peuvent être facilement vérifiées par un moteur de réécriture standard.

4.1 La Sémantique de la logique de réécriture de Java

Syntaxe La spécification de la sémantique opérationnelle de Java est une théorie de réécriture composée de 424 équations et 7 règles. Elle est représentée par le triplet :

$$\Re_{Java} = (\Sigma_{Java}, E_{Java}, R_{Java})$$

Où :

- Σ_{Java} est une signature triée représentant la structure statique du programme Java
- $E_{Java} = \Delta_{Java} \uplus B_{Java}$ est un ensemble d'axioms équationnels, avec B_{Java} des axioms tel que l'associativité, la commutativité et l'unité, et Δ_{Java} l'ensemble des équations déterministes de Σ_{Java}
- R_{Java} est un ensemble de règles de réécriture de Σ_{Java} représentant les caractéristiques concurrentes du programme Java.

Règles de Réécriture Soit u et v des états du programme, et $r \in R_{java} \cup \Delta_{Java}$. La représentation

$$u \rightarrow^{r}_{Java} v$$

implique que u est **réécrit** en v en utilisant r.

L'extension à cette représentation à plusieurs étapes de réécriture est notée :

$$u \rightarrow^{*}_{Java} v$$

Théorème 5. $u \rightarrow^{*}_{Java} v$ ***ssi*** *il existe* $u_1, \ldots, u_k$ *tel que*

$$u \rightarrow_{Java} u_1 \rightarrow_{Java} u_2 \ldots u_k \rightarrow_{Java} v$$

Types : *State* et *Value* La théorie de réécriture $\Re_{Java}$ est définie comme ensemble de termes d'un état concret *State* avec les attributs *in, out, mem* et *store*. Ils définissent une structure paramétrique par rapport à un type générique *Value*, qui définit toutes les valeurs possibles retournées par les fonctions Java ou sauvegardées en mémoire.

Les continuations La sémantique de Java utilise le principe des **continuations**. Les continuations permettent de maintenir le contrôle du contexte de chaque processus tout en spécifiant explicitement la prochaine étape à réaliser pas le processus.

Soit la continuation : $e \longrightarrow k$. Une fois l'expression e évaluée, son résultat est passé au reste de la continuation (k). Prenons par exemple l'opération d'addition en Java. Elle est représentée par des continuations comme suit :

```
eq k((E + E') -> K ) = k((E,E') -> (+ -> K)) .
eq k((int(I), int(I')) -> (+ -> K)) = k(int(I + I') -> K) .
```

k représente un symbole montrant une continuation dans un processus. **->** permet de concaténer les continuations et l'opérateur + est fournis sous deux formes : avec une arité égale à 2, pour représenter l'addition dans Maude, et avec une arité égale à 0 pour représenter un symbole de continuation utilisé pour empiler l'action d'addition.

Nous montrons également dans l'exemple suivant l'instruction *if/then/else*, qui sera modifiée par la suite pour intégrer le contrôle de flux d'information :

```
eq k((if E S else S' fi) -> K) = k(E -> (if(S,S') -> K) .
eq k(bool(true) -> (if(S,S') -> K)) = k(S -> K) .
eq k(bool(false) -> (if(S,S') -> K)) = k(S' -> K) .
```

4.2 Logique de réécriture pour le contrôle de flux d'information de Java

JML : Langage de spécification des niveaux de sécurité Pour attribuer les annotations de sécurité aux éléments d'un programme en Java, [Alba-Castro09] utilise JML (*Java Modeling Language*), un langage de spécification d'interface permettant de décrire des informations statiques dans les modules Java en utilisant des **préconditions** (grâce à la clause *requires*), des **postconditions** (clause *ensures*), des **invariants** (clause *invariant*) et des **assertions** (clause *assert*).

Les classes de sécurité dans le programme sont affectées aux éléments de stockage de manière statique, en utilisant les clauses décrites ci-dessus de la manière suivante :

- La clause **ensures** est utilisée pour indiquer le niveau de confidentialité attendu par l'utilisateur d'une méthode.
- La clause **requires** permet d'indiquer les niveaux de confidentialité des paramètres d'entrée d'une méthode.
- La clause **assert** permet d'indiquer les niveaux de confidentialité des variables locales.

La non-interférence La non-interférence est la propriété de sécurité ciblée dans ce travail. Le but des outils décrits dans [Alba-Castro09] est de vérifier que les variables avec un niveau de confidentialité fixé n'influent pas sur les sorties de niveau de confidentialité plus bas.

La non-interférence est représentée par la relation $< L, \subseteq >$, avec L l'ensemble des étiquettes et $\subseteq$ la relation "*peut circuler vers*" ; ainsi que la fonction d'étiquetage $Lab : Var \rightarrow L$ qui associe à une variable dans Var une étiquette dans L.

Les niveaux de confidentialité utilisés sont Low et $High$ avec $Low \subseteq High$. On définit également l'opérateur de jointure $\bigcup$ comme suit :

- $Low \bigcup Low = Low$
- $X \bigcup Y = High$ sinon.

Le modèle d'étiquettes utilisé dans ces travaux est un modèle simplifié, qu'on peut facilement généraliser au modèle d'étiquettes désiré en redéfinissant la relation $\subseteq$ et la relation $\bigcup$ entre les différentes étiquettes, pouvant ainsi englober plusieurs niveaux de sécurité, ainsi que la propriété d'intégrité. Dans l'annexe C, partie IV, nous présentons une proposition de définition du type Label pour le modèle d'étiquettes à jetons présenté dans la section III.2.2, partie I.

Nous présentons dans ce qui suit deux exemples de flux d'information illégal représenté dans un programme simple. Le premier représente un flux explicite (une affectation) et le second un flux implicite (un bloc *if*).

Exemple (Flux explicite illégal).

```
public int mE1 (int h, int l){
            l = h;
            return l;
}
/*@ requires h == High && l == Low;
  @ ensures \result == Low;
@*/
```

Exemple (Flux implicite illégal).

```
public int mE2 (int h, int l){
            if (h > 2) l = 0;
            return l;
}
/*@ requires h == High && l == Low;
  @ ensures \result == Low;
@*/
```

4.3 Sémantique de Java étendue au contrôle de flux d'information

Pour représenter le contrôle de flux d'information, la sémantique de Java doit être étendue de manière à attribuer à une donnée son étiquette aussi bien que sa valeur. Le domaine *Value* va ainsi être étendu à *Value* x *LValue*, avec *LValue* représentant l'ensemble des étiquettes possibles dans le modèle, soit dans ce cas les valeurs *High* et *Low*. Ainsi, on associe à une variable le couple $< Value, LValue >$ pour représenter à la fois sa valeur et son niveau de sécurité. On définit également le constructeur *lenv* permettant de sauvegarder le niveau global de sécurité. Il permet ainsi de maintenir le compteur programme (*pc*) à jour.

L'exemple suivant montre comment est-ce que l'instruction *if/then/else* a été réécrite pour ajouter le contrôle de flux d'information.

Exemple (Instruction *if/then/else* réécrite).

```
-- Évaluer l'expression booléenne
-- et garder l'étiquette du contexte courant
eq k((if E S else S' fi) -> K) lenv(LEnv)
  = k(E -> if(S, S') -> restoreLEnv(LEnv) -> K) lenv(LEnv) .
-- Mettre à jour l'étiquette du contexte
eq k(< bool(true), LValue > -> (if(S, S') -> K)) lenv(LEnv)
  = k(S -> K) lenv(LEnv join LValue) .
eq k(< bool(false), LValue> -> (if(S, S') -> K)) lenv(LEnv)
  = k(S' -> K) lenv(LEnv join LValue) .
-- Restaurer l'ancienne étiquette de contexte
eq k(restoreLEnv(LEnv) -> K) lenv(LEnv') = k(K) lenv(LEnv) .
```

4.4 Sémantique de réécriture abstraite de Java

Dans le but de vérifier des systèmes volumineux, on définit des *abstractions*. L'idée est de considérer un système fini qui abstrait les caractéristiques du système infini initial auquel nous nous intéressons, de transformer la propriété que nous désirons prouver dans ce nouveau système et d'y appliquer la vérification par réécriture. Le système abstrait devrait être suffisamment explicite pour que la propriété s'y applique, et en même temps d'une taille restreinte pour être couvert par la vérification.

[Alba-Castro09] définit une abstraction de la logique de réécriture de Java par :

$$\Re_{Java\#} = (\Sigma_{Java\#}, E_{Java\#}, R_{Java\#})$$

où : $E_{Java^{\#}} = \Delta_{Java^{\#}} \uplus B_{Java^{\#}}$

La relation de réécriture associée est alors symbolisée par : $\rightarrow_{Java^{\#}}$

Cette représentation est une extension de la théorie de réécriture originelle $\Re_{Java}$ avec une abstraction du domaine de valeurs à `LValue = {Low,High}`, ce qui veut dire que les vraies valeurs des données sont négligées. Grâce à cette abstraction, plusieurs pas de calcul de $\longrightarrow_{Java^{E}}$ sont remplacés par un seul pas de calcul abstrait de $\longrightarrow_{Java^{\#}}$, ce qui reflète le fait que plusieurs comportements distincts sont compressés en un seul état abstrait.

La sémantique de réécriture de Java étendue aux flux d'information définie dans la sous-section section III.4.3, partie IV est abstraite de manière à ce que :

1. Chaque paire $< Value, LValue >$ dans les équations et règles est approximée par le deuxième élément $LValue$
2. Les équations ne pouvant pas être prouvées confluentes [2] après la transformation sont transformées en règles pour refléter les réécritures possibles d'un état abstrait.

4.5 Certification

La méthodologie décrite par [Alba-Castro09] permet de générer un **certificat de sûreté** qui consiste en un ensemble de preuves de réécritures abstraites de la forme :

$$t_1 \longrightarrow^{r_1}_{Java^{\#}} t_2 \ldots \longrightarrow^{r_{k-1}}_{Java^{\#}} t_k$$

Elles décrivent implicitement les états du programme qui peuvent être atteints à partir d'un état initial donné. Puisque ces preuves correspondent à l'exécution de la sémantique abstraite de Java, qui est fournie au programmeur, le certificat peut être vérifié par n'importe quel moteur de réécriture standard. Il suffit de vérifier que chaque pas de réécriture abstrait dans le certificat est valide et qu'aucune chaîne n'a été négligée.

Conclusion

Les systèmes de réécriture peuvent être utilisés dans notre plateforme pour vérifier la propagation des étiquettes dans le code de chaque composant. Grâce au certificat de sûreté, les différentes étapes utilisées pour la vérification peuvent être suivies.

2. La **confluence** de deux chemins implique que, même s'ils divergent d'un ancêtre commun, ces chemins finissent par se joindre à un successeur commun.

Conclusion et Perspectives

La vie est l'art de tirer des conclusions
suffisantes de prémisses insuffisantes.
[Samuel Butler]

Conclusion Générale et Perspectives

L'application de la non-interférence sur un système distribué à un niveau de granularité fin demeure difficile, et ce à cause de la dynamicité de ces systèmes, de leurs architecture complexe, de l'hétérogénéité de leurs nœuds, leur utilisation de modules patrimoniaux qui peuvent être présentés sous la forme de boîtes noires... De plus, la vérification de la non-interférence requiert un contrôle de flux d'information, qui n'est pas toujours évident pour les systèmes volumineux, et qui peut s'avérer ardu à utiliser pour un développeur.

Nous avons cité plusieurs travaux de l'état de l'art qui se rapprochent de notre thématique et les avons répertorié en deux catégories : les solutions offrant des modules indépendants pour assurer la sécurité dynamique des systèmes distribués, et les solutions appliquant le contrôle de flux d'information aux applications.

Les premières solutions proposent des modules qui gèrent la sécurité du système réparti à l'exécution. Tout comme DCIF, ces modules ont la particularité de séparer l'implémentation de la sécurité du code fonctionnel et permettent l'application et la vérification dynamique de la politique spécifiée à la compilation. Cependant, aucun d'entre eux ne permet de faire un contrôle du flux d'information du système.

La deuxième catégorie de solutions offre plusieurs intergiciels pour appliquer le contrôle de flux d'information sur les systèmes. Nous les avons répertorié en solutions statiques et solutions dynamiques.

Solutions statiques Les solutions statiques proposent de réaliser une analyse de flux d'information, qui permet d'inférer les dépendances des informations entre les variables du programme, et vérifier ainsi que le programme ne contient pas de flux d'information illégaux. La solution principale sur laquelle nous nous sommes basés est JIF, qui propose un système de type et un compilateur pour écrire un code non-interférent à la compilation.

L'inconvénient principal de JIF est que le concepteur de l'application doit écrire tout le code de son application en utilisant les annotations. Cela implique principalement deux choses : que les contraintes de sécurité sont entremêlées avec les contraintes fonctionnelles et présentées dans le même code, mais également que le concepteur a la tâche ardue d'attribuer des étiquettes de sécurité à toutes les variables de son système, même les variables intermédiaires, et faire en sorte que ses annotations respectent la politique de sécurité désirée sans être trop restrictives ni trop lâches.

Les outils que nous définissons permettent de remédier à ces inconvénients en fournissant au concepteur de l'application (1) un modèle lui permettant de décrire la politique de sécurité de son système à un haut niveau d'abstraction en affectant les étiquettes de sécurité uniquement aux interfaces des composants, et (2) un outil permettant de propager ces étiquettes dans le code, pour les attribuer aux variables intermédiaires et vérifier que la configuration initiale est non-interférente.

Une autre solution présente un intérêt à nos yeux : c'est FlowCaml. Ce système simplifie grandement l'application du contrôle de flux d'information au système en utilisant l'inférence de type pour automatiser la propagation des étiquettes. Mais, contrairement à notre approche, le langage utilisé est un langage fonctionnel pour des systèmes centralisés, et l'annotation manuelle est faite à un niveau d'abstraction plus bas que le nôtre, puisque le programmeur doit annoter les différentes entrées des fonctions, alors que nous réalisons l'annotation au niveau des composants (représentés par une classe, donc un ensemble de fonctions), pour ensuite la propager aux différents objets et variables définis.

Plusieurs solutions statiques sont également proposées pour les systèmes distribués, certaines sont des extensions de JIF, tel que JIF/Split et Fabric. L'inconvénient principal de JIF/Split −outre le fait que l'annotation du programme initial est faite à la main− est que le partitionnement de l'application est fait en suivant des contraintes de sécurité, contrairement à notre approche qui démarre avec un code initialement partitionné selon les contraintes fonctionnelles de l'architecte du système, puis applique dessus la politique de sécurité. Fabric, de son côté, exige que les systèmes doivent être entièrement écrits dans le langage Fabric et ne supportent pas l'utilisation de composants patrimoniaux, ce qui rend l'application de cette solution aux systèmes existants assez ardue. Pour ces deux solutions, les opérations de cryptographie qui permettent d'assurer la communication entre les différents nœuds qui exécutent les sous-programmes ne sont pas gérées et doivent donc être prises en main par le développeur lui-même.

Le compilateur de Fournet résout ce problème car les opérations de cryptographie sont prises en compte automatiquement par le système. Néanmoins, là également, le partitionnement du système en nœuds est fait en fonction des étiquettes de sécurité, ce qui pourrait ne pas

correspondre aux besoins fonctionnels de l'architecte du système. De plus, tout le code est annoté manuellement.

D'autre part, la solution offerte par [Alpizar09] est intéressante pour prouver la non-interférence dans les systèmes répartis utilisant un langage similaire au leur. Cependant, ce langage est simpliste, et ne couvre pas tous les cas complexes que nous trouvons dans un programme écrit en Java. De plus, ce système n'est pas implémenté.

Solutions dynamiques Certaines solutions dynamiques proposent d'appliquer la non-interférence au niveau du système d'exploitation. Ces travaux peuvent être complémentaires au nôtre. En effet, nous faisons l'hypothèse que l'infrastructure est sécurisée (système, machines, etc) et nous focalisons sur la sécurisation des applications. Les solutions précédentes, en contrepartie, ciblent la sécurisation de l'infrastructure et laissent la responsabilité de la sécurité du code applicatif au développeur, elles peuvent donc représenter des bases fiables sur lesquelles appliquer notre intergiciel. Cependant, elles engendrent des problèmes de performance et sont compliquées à généraliser à d'autres types de systèmes d'exploitation que ceux considérés. Dans notre travail, l'utilisateur choisit la granularité avec laquelle il considère le contrôle de flux d'information ; par exemple, seuls les ports d'entrée/sortie sont considérés pour les composants patrimoniaux de confiance et non pas la totalité du code. Cette caractéristique peut aider à améliorer considérablement la performance du système puisque le contrôle de flux d'information est utilisé seulement si nécessaire. De plus, nous renforçons le contrôle de flux d'information par des mécanismes cryptographiques entre les composants distants.

D'autres solutions de contrôle de flux d'information dynamique sont présentées, notamment DStar qui assure principalement la communication sécurisée et sans interférences entre des processus sur des nœuds différents, ce qui équivaut plus généralement à la partie "vérification inter-processus" de notre travail. De plus, sur une même machine, un même exporteur est utilisé pour toute communication avec le monde extérieur, ce qui est beaucoup moins flexible que l'utilisation de ports spécifiques et distincts, portant chacun une étiquette bien définie restreignant sa communication.

Nous présentons également quelques solutions qui s'appliquent à des types de systèmes particuliers, comme les systèmes à base d'évènements pour SmartFlow, ou des protocoles de communication particuliers comme les services web. Notre travail s'abstrait de ces systèmes pour créer une solution générique pour tous types de systèmes à base de composants distincts.

CIF Au vu de ces solutions, nous avons construit CIF, qui offre un moyen simple et pratique d'attribuer des étiquettes de sécurité au système à un haut niveau d'abstraction grâce à un fichier *Policy*, et qui offre un ensemble d'outils permettant de saisir ces informations à partir de ce fichier, de l'appliquer au système distribué construit au préalable, et de réaliser la propagation

des étiquettes dans le code de chaque composant. D'autre part, la non-interférence est vérifiée entre les composants en comparant les étiquettes d'entrée et de sortie, et les composants de cryptographie sont automatiquement générés.

Les avantages de CIF sont principalement :

- **La simplicité** : CIF permet à l'administrateur d'assigner des étiquettes aux entrées et sorties des composants, et donc à un haut niveau d'abstraction, grâce à un fichier XML qui ne demande aucun effort d'apprentissage particulier.
- **La généricité** : CIF peut être appliqué à tout système distribué à base de composants. L'utilisation du paradigme orienté composants est certes un grand avantage pour le contrôle de flux, un problème pourrait se poser néanmoins pour les applications distribuées existantes qui ne l'utilisent pas. Dans ce cas, il est possible d'abstraire ces systèmes de manière à transformer leur architecture en une architecture explicite à base de composants [Abdellatif06]. De plus, CIF peut être appliqué à tout modèle orienté composants, pour tout modèle d'étiquettes (respectant les conditions énumérées dans la section I.2, partie III).
- **La flexibilité** : CIF offre des mécanismes simples permettant d'étendre le compilateur initial pour intégrer de nouveaux modèles d'étiquettes ou de nouveaux modèles d'architecture dans le code initial de CIF, et ce sans mettre en péril son comportement principal.
- **L'automatisation** : CIF permet de propager les étiquettes dans le code à partir d'une description de politique à un niveau très abstrait. La propagation est quasi-automatique et permet d'épargner au développeur la difficulté d'écrire son code dans un langage typé-sécurité particulier.
- **La réutilisation** : CIF offre une solution pour gérer les composants patrimoniaux. Cette solution garantit la vérification de la non-interférence dans le code du composant sans porter atteinte à la confidentialité de son code. D'autre part, dans le fichier *Policy*, il est possible de configurer à haut niveau toute bibliothèque importée et spécifier ainsi ses propriétés de sécurité sans avoir à annoter son code en entier.

Cependant, les outils CIF peuvent encore être étendus, pour intégrer d'autres aspects. L'outil CIFIntra ne supporte pas encore les threads, les classes imbriquées et les exceptions non déclarées. D'autre part, le module contrôleur, qui (pour l'instant) autorise une information à être rétrogradée en considérant les capacités statiques des composants, pourrait en plus vérifier que cette rétrogradation ne va pas nuire à la sécurité globale du système [Sabelfeld09] en vérifiant par exemple la propriété de robustesse [Chong06].

Les composants de cryptographie générés implémentent pour l'instant uniquement le chiffrement asymétrique et la signature MD5. Une amélioration possible de ces travaux est de donner

à l'administrateur le choix de l'algorithme de cryptographie à utiliser, le type des opérations à réaliser...

DCIF Le canevas DCIF propose d'étendre les outils CIF pour réaliser le contrôle de flux à l'exécution. En effet, tout changement dans l'architecture du système pendant l'exécution peut influer sur la politique de sécurité, dans le sens où il perturbe le flux d'information préalablement vérifié. Il est donc important de refaire un contrôle à chaque remplacement de composant, établissement d'une liaison...

Le canevas DCIF peut être très utile pour gérer l'impact des pannes sur le système. En effet, les systèmes distribués ont la vulnérabilité de leur composant le plus faible. Il est donc important, pour les systèmes critiques en particulier, de fournir un mécanisme de remplacement de composants. Ce remplacement peut se faire à tout moment de l'exécution, ce qui peut provoquer des interférences qui n'étaient pas prévues à la compilation.

Le canevas DCIF comporte un ensemble de composants de sécurité qui interagissent pour assurer la sécurisation du système. Ses principaux avantages sont :

- **La transparence** : Toute l'opération de reconfiguration de la sécurité est transparente à l'utilisateur. C'est le canevas qui, avant de remplacer ou de modifier des composants ou des liaisons, procède à la vérification de leur impact sur le reste du système, sans notifier l'utilisateur, sauf en cas de détection d'interférence qui ne peut pas être gérée par le contrôleur.
- **La modularité** : DCIF est un canevas modulaire qui respecte le paradigme orienté composants. Il permet de fournir un ensemble de composants de sécurité au préalables, et d'en générer d'autres selon les besoins (en particulier les composants de cryptographie, à la manière de CIF).
- **L'automatisation** : DCIF permet d'automatiser le processus de vérification en utilisant des déclencheurs qui détectent tout changement d'architecture (les composants *Factory*) et un composant qui orchestre la communication entre les composants de sécurité (le *Security Manager*).

L'une des perspectives de notre travail est de mettre en œuvre le canevas DCIF dans un environnement distribué réel en respectant la description définie dans le chapitre II, partie III. Pour plus de sécurité, l'implantation initiale des clefs de cryptographie pour les différents domaines et composants peut se faire en utilisant, par exemple, des composants cryptographiques matériels (TPM pour *Trusted Platform Module*) [3].

3. **TPM** : https ://www.trustedcomputinggroup.org/

D'autre part, une preuve formelle plus élaborée peut être réalisée pour prouver que la totalité du système produit pas CIF est non-interférent, en étendant, par exemple, la solution présentée par [Alpizar09] aux systèmes écrits dans un langage orienté objet.

Notre travail peut être appliqué sur des systèmes distribués volumineux et critiques, où toute fuite d'information peut être fatale, tel que par exemple les systèmes de vote électronique[Clarkson08], qui ont la particularité d'être hautement dynamiques et dont toute fuite peut influer sur la propriété d'anonymat des participants[Moskowitz03, Chatzikokolakis08].

Références Bibliographiques

[Abdellatif06] T. Abdellatif. *Apport des architectures à composants pour l'administration des intergiciels*. PhD thesis, 2006. 124

[Abdellatif07] T. Abdellatif, J. Kornas, and J.B. Stefani. Reengineering J2EE Servers for Automated Management in Distributed Environments. *Distributed Systems Online, IEEE*, 8, November 2007. 19

[Abdellatif11] T. Abdellatif, L. Sfaxi, R. Robbana, and Y. Lakhnech. Automating Information Flow Control in Component-based Distributed Systems. *International Symposium on Component-based System Engineering, CBSE*, 2011. 8

[Alba-Castro09] M. Alba-Castro, M. Alpuente, and S. Escobar. Automated certification of noninterference in rewriting logic. *Formal Methods for Industrial Critical Systems*, pages 182–198, 2009. 108, 109, 110, 114, 116, 117, 118, 119

[Alba-Castro10] M. Alba-Castro, M. Alpuente, and S. Escobar. Abstract Certification of Global Non-Interference in Rewriting Logic. *Arxiv preprint arXiv :1006.4304*, pages 1–20, 2010. 109

[Alpizar09] R. Alpizar and G. Smith. Secure Information Flow for Distributed Systems. In *Formal Aspects in Security and Trust : 6th International Workshop, Fast 2009, Eindhoven, the Netherlands, November 5-6, 2009, Revised Selected Papers*, page 126, 2009. iii, 54, 123, 126

[Amtoft06] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, number 1, pages 91–102. ACM, January 2006. 27, 71

[Basin06] D. Basin and J. Doser. Model driven security : From UML models to access control infrastructures. *ACM Transactions on Software*, 15 :39–91, 2006. 40

[Beisiegel05] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, and J. Marino. SCA service component architecture-assembly model specification. *SCA Version 0.9, November*, 2005. 19

[Beisiegel07] M. Beisiegel, D. Booz, C.Y. Chao, M. Edwards, A. Karmarkar, S. Lelceanu, A. Malhotra, E. Newcomer, S. Patil, M. Rowley, and C. Sharp. SCA Policy Framework. Technical report, IBM, Oracle, IONA, SAP, BEA, TIBCO, 2007. 24

[Bell75] D.E. Bell and L.J. LaPadula. Secure computer system : Unified exposition and Multics interpretation. *Technical Report n. ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA*, August 1975. 3

[Biba77] K.J. Biba. Integrity Considerations for Secure Computer Systems - Storming Media. *Technical Report. MITRE CORP BEDFORD MA*, 1977. 3

[Blaze99] M. Blaze, J. Feigenbaum, and J. Ioannidis. The role of trust management in distributed systems security. *Secure Internet*, 1999. 42, 43

[Broy98] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component ? *Software-Concepts & Tools*, 19(1) :49–56, June 1998. 19

[Chatzikokolakis08] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4) :378–401, February 2008. 126

[Chmielewski08] L. Chmielewski, R. Brinkman, J.H. Hoepman, and B. Bos. Using JASON to secure SOA. In *Proceedings of the 2008 workshop on Middleware security*, pages 13–18, New York, New York, USA, 2008. ACM. 41

[Chong06] S. Chong and A.C. Myers. Decentralized robustness. *19th IEEE Computer Security Foundations Workshop,*, pages 242–256, 2006. 52, 124

[Chong07] S. Chong, K. Vikram, and A.C. Myers. SIF : Enforcing confidentiality and integrity in web applications. *Proceedings of 16th USENIX*, 2007. 52

[Clarkson08] M.R. Clarkson, S. Chong, and A.C. Myers. Civitas : Toward a Secure Voting System. *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 354–368, May 2008. 126

[Clavel07] M. Clavel, S. Eker, F. Durán, P. Lincoln, N. Mart\'\i-Oliet, and J. Meseguer. *All about Maude : A High-performance Logical Framework : how to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350. Springer-Verlag New York Inc, 2007. 110

[Denker98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols*. Citeseer, 1998. 109

[Denning77] D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, (7) :504–513, July 1977. 3, 55

[Dershowitz89] N. Dershowitz and J.P. Jouannaud. *Rewrite systems*. May 1989. 108

[Efstathopoulos05] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, volume 25, page 30. ACM, 2005. 3, 56

[Eyers09] D. M. Eyers, B. Roberts, J. Bacon, I. Papagiannis, M. Migliavacca, P. Pietzuch, and B. Shand. Event-processing middleware with information flow control. In *Middleware '09 : Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–2, New York, NY, USA, 2009. Springer-Verlag New York, Inc. 56

[Fassino02] J.P. Fassino, J.B. Stefani, J.L. Lawall, and G. Muller. THINK : A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002. 44

[Fournet09] C. Fournet, G. Le Guernic, and T. Rezk. A Security-Preserving Compiler for Distributed Programs. *Proceedings of the 16th ACM conference on Computer and communications security*, pages 432–441, 2009. iii, 53, 59

[Genet09] T. Genet. *Reachability analysis of rewriting for software verification*. PhD thesis, 2009. 110

[Goguen82] J.A. Goguen and J. Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, page 11, 1982. 2, 3, 12, 25

[Heiser05] G. Heiser. Secure embedded systems need microkernels. *USENIX ; login*, 30 :9–13, 2005. 45

[Hunt06] S. Hunt and D. Sands. On flow-sensitive security types. In *ACM SIGPLAN Notices*, number 1, pages 79–90. ACM, January 2006. 71

[Hutter06] D. Hutter and M. Volkamer. Information flow control to secure dynamic web service composition. *Security in Pervasive Computing*, 2006. 36, 57, 91

[Khair98] M. Khair and I. Mavridis. Design of secure distributed medical database systems. *Database and Expert Systems*, 1998. 29, 34

[Klop92] J.W. Klop. Term rewriting systems. *Handbook of logic in computer science*, (3) :1–116, May 1992. 108

[Krohn07] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M.F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review*, 2007. 3, 29, 31, 56

[Kuz07] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES : A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80 :687–699, May 2007. 45

[Lacoste08] M. Lacoste, T. Jarboui, and R. He. A component-based policy-neutral architecture for kernel-level access control. *Annals of telecommunications - annales des télécommunications*, 64 :121–146, November 2008. 44

[Li07] G. Li and M. Ogawa. On-the-fly model checking of fair non-repudiation protocols. In *Proceedings of the 5th international conference on Automated technology for verification and analysis*, pages 511–522. Springer-Verlag, October 2007. 109

[Liu09] J. Liu, M.D. George, K. Vikram, X. Qi, L. Waye, and A.C. Myers. Fabric : A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 321–334. ACM, 2009. 53

[Malecha10] G. Malecha and S. Chong. A more precise security type system for dynamic security tests. *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security - PLAS '10*, pages 1–12, 2010. 25

[Meseguer92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96 :73–155, April 1992. 108

[Moskowitz03] I.S. Moskowitz, R.E. Newman, D.P. Crepeau, and A.R. Miller. Covert channels and anonymizing networks. In *Proceeding of the ACM workshop on Privacy in the electronic society - WPES '03*, page 79, New York, New York, USA, 2003. ACM Press. 126

[Myers00] A.C. Myers and B Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2000. 3, 29, 49, 104

[Myers06] A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2) :157–196, 2006. 25, 52, 84, 104

[Myers99] A.C. Myers. JFlow : Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, number January, pages 228–241. ACM, 1999. 49

[Nikander99] P. Nikander. *An architecture for authorization and delegation in distributed object-oriented agent systems.* Citeseer, 1999. 13, 17, 41, 43

[Nystrom03] N. Nystrom, M.R. Clarkson, and A.C. Myers. Polyglot : An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, April 2003. 50, 67

[Ogata04] K. Ogata and K. Futatsugi. Rewriting-based verification of authentication protocols. *Electronic Notes in Theoretical Computer Science*, 71 :208–222, April 2004. 109

[Rushby92] J. Rushby. Noninterference, transitivity, and channel-control security policies. *Technical Report No. CSL-92-02.*, (650), 1992. 25

[Russo10] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 186–199. IEEE, 2010. 71

[Sabelfeld09] A. Sabelfeld and D. Sands. Declassification : Dimensions and principles. *Journal of Computer Security*, 17(5) :517–548, 2009. 71, 124

[Schneider00] F.B. Schneider. Enforceable security policies. *Transactions on Information and System Security (TISSEC)*, 3 :30–50, 2000. 27

[Seinturier09] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.B. Stefani. Reconfigurable SCA applications with the frascati platform. In *2009 IEEE International Conference on Services Computing*, volume 0, pages 268–275. IEEE, 2009. 24

[Seitz03] L. Seitz, J.-M. Pierson, and L. Brunie. Key management for encrypted data storage in distributed systems. *Second IEEE International Security in Storage Workshop*, pages 20–20, 2003. 45

[Sfaxi10] L. Sfaxi, Takoua Abdellatif, Y. Lakhnech, and R. Robbana. Contrôle du flux d'information des systèmes distribués à base de composants. In *10ème Conférence Internationale sur les NOuvelles Technologies de la REpartition, NOTERE*, pages 189–196, Tozeur, Tunisia, 2010. 8

[Sfaxi11] L. Sfaxi, T. Abdellatif, R. Robbana, and Y. Lakhnech. Information Flow Control of Component-based Distributed Systems. *Concurrency and Computation, Practice and Experience, Wiley*, pages 1–6, 2011. 8

[Sfaxi11a] L. Sfaxi, T. Abdellatif, Y. Lakhnech, and R. Robbana. Sécuriser les Systèmes Distribués à base de Composants par Contrôle de Flux d'Information. *Technique et Science Informatiques (TSI)*, 30 :1–35, 2011. 8

[Simonet03] V. Simonet. The Flow Caml System. Technical report, INRIA, 2003. 49, 52

[Welch03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid services. In *Proceedings. 12th IEEE International Symposium on High Performance Distributed Computing, 2003.*, pages 48–57. IEEE Comput. Soc, 2003. 17, 41, 42, 46

[Zdancewic02] S. Zdancewic, L. Zheng, N. Nystrom, and A.C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20 :328, August 2002. 52, 53

[Zdancewic03] S. Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science*, pages 1–16. Citeseer, 2003. 4, 52

[Zdancewic04] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID-04)*. Citeseer, 2004. 3, 37

[Zeldovich06] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, volume pages, page 278. USENIX Association, 2006. vi, 3, 31, 33, 56

[Zeldovich08] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308. USENIX Association, 2008. 33, 56

Table des Abbréviations

ACL	Access Control List, 13
ADL	Architecture Description Language, 19
ADT	Abstract Data Type, 110
AES	Advanced Encryption Standard, 14
API	Application Programming Interface, 65
AST	Abstract Syntax Tree, 67
CA	Certificate Authority, 16
CAS	Community Authorization Service, 42
CBSE	Component-Based Software Engineering, 4, 19
CFI	Contrôle de Flux d'Information, 3, 25
CIF	Component Information Flow, 5, 61
CIFInter	CIF for Inter-component verification, 73
CIFIntra	CIF for Intra-component verification, 67
CIForm	CIF Intermediate Format, 65
CM	Crypto Manager (DCIF), 78
DCIF	Dynamic Component Information Flow, 5, 76
DDoS	Distributed Denial of Service, 11
DES	Data Encryption Standard, 14
DIFC	Distributed Information Flow Control, 55
DKMS	Distributed Key Management System, 45

DLM	Decentralized Label Model, 29
DoS	Denial of Service, 11
EJB	Enterprise Java Bean, 40
GM	Global Manager (DCIF), 76
IBA	Internal Binding Artifact, 72
IBP	Internal Binding Plotter, 72
IFCM	Information Flow Control Manager (DCIF), 77
IKE	Internet Key Exchange, 41
ISAKMP	Internet Security And Key Management Protocol, 41
ITSEC	Information Technology Security Evaluation Criteria, 6
JCE	Java Cryptography Extension, 154
JIF	Java Information Flow, 49
JML	Java Modeling Language, 116
JSSE	Java Secure Sockets Extension, 154
KINK	Kerberized Internet Negotiation of Keys, 41
KM	Key Manager (DCIF), 77
MD5	Message Digest 5, 13
MDS	Model Driven Security, 40
MITM	Man in the Middle, 11
PBE	Password Based Encryption, 15
pc	Program Counter, 50, 68
RBAC	Role-Based Access Control, 40
RSA	Rivest Shamir Adleman, 15

SCA	Service Component Architecture, 22
SHA-1	Secure Hash Algorithm-1, 14
SM	Security Manager (DCIF), 78
SOA	Service-Oriented Architecture, 22
SPKI	Simple Public Key Infrastructure, 17, 44
TPM	Trusted Platform Module, 125
TRS	Term Rewriting System, 108
URH	User Role Hierarchy, 34
VO	Virtual Organization, 42

Annexes

Annexe A

Algorithme de "Internal Binding"

Types de variables

- Entrées de composants (**CI** pour *Component Inputs*)
  - Entrées des composants fonctionnels
  - Représentés par les attributs de composants (attributs Java) et les ports d'entrées (méthodes Java).
- Sorties de composants (**CO** pour *Component Outputs*)
  - Sorties des composants fonctionnels
  - Représentés par les ports de sortie (attributs Java).
- Variables intermédiaires
  - Toute autre variable dans le code.

⇒ À la fin de l'exécution de cet algorithme, seules les dépendances des COs par rapport aux CIs sont gardées.

Compteur Programme

Le *pc* (pour *Program Counter*) est calculé pour chaque ligne de code. Il représente l'ensemble de variables desquelles l'exécution de l'instruction en cours dépend. Il est utilisé pour éviter les flux implicites.

- Initialement, le *pc* est mis à *{}*.

- Dans la définition de chaque méthode, la valeur initiale du *pc* est déterminée selon le contexte où elle est appelée. Sa valeur initiale est donc variable et elle dépend de la valeur du *pc* du contexte appelant. On la nomme *pcm*.
- Pour toute instruction qui n'est pas contenue dans un bloc, la valeur du *pc* ne change pas.
- Si une instruction est conditionnelle ou itérative, la valeur du *pc* est déterminée par la réunion des variables de l'expression conditionnelle ou itérative.

Détermination de la relation de dépendance

La relation de dépendance est calculée selon le tableau A.1. Il est à noter que :

- *depend* est une fonction transitive :

$$\textbf{Si} \;\; (x = depend(y)) \;\; \textbf{et} \;\; (y = depend(z)) \;\; \textbf{alors} \;\; x = depend(z)$$

- *ModifVars(Expr)* représente les variables modifiées dans l'expression *Expr*. Ces variables sont utilisées dans une affectation (comme opérande gauche) et dans les appels de méthode (comme objets appelants).
- Si une variable dépend d'une expression, elle dépend de toutes les variables de cette expression.
- En quittant un bloc ou une méthode, toutes les dépendances des variables locales à ce bloc sont calculées et seules les variables définies à l'extérieur du bloc sont gardées. Par exemple, en quittant la définition d'une méthode, seules les dépendances des variables de retour, des CO et des CI sont gardées. Les variables locales intermédiaires sont éliminées grâce à la fonction de transition définie plus haut. Par exemple. Dans la méthode :

```
int meth(boolean a){
    boolean b =!a;
    return b;
}
```

  on déduit que :
  - b=depend(*pcm*,a)
  - return(meth) = depend(*pcm*,b)

  Ce qui implique que : return(meth)=depend(*pcm*,a).
- Les relations de dépendance d'une méthode donnée sont calculées une seule fois. L'algorithme calcule les dépendances entre les variables de retour et/ou les CO, et les paramètres d'entrée formels et/ou les CI. Prenons par exemple la méthode suivante :

```
int meth(boolean a){
    boolean b =!a;
    return b;
}
```

- À la sortie de la méthode, nous avons : return(meth) = depend(*pcm*,a)
- À l'appel de la méthode :

```
boolean i = true;
boolean j;
int v1,v2;
...
if ( v1 == v2 ){
    j=meth(i);
}
```

Au niveau de l'instruction $j = meth(i)$, la valeur du *pc* est mise à *{v1,v2}* et le paramètre formel a est remplacé par le paramètre effectif i. Dans ce cas, *pcm* prend la valeur du *pc*, et $a = depend(i)$. Ce qui donne :

$$j = depend(pc, i) = depend(v1, v2, i)$$

Tableau A.1 – La relation de dépendance pour le calcul des IB

Flux Explicites			
Affectation ou initialisation	x=y Type x = y	x=depend(y)	Dans une affectation, la valeur de l'opérande de gauche dépend des opérandes de droite
Appel de méthode	Type x = meth(y)	x=depend(y) x=depend(return(meth)) pcm(meth)=pc	Dans un appel de méthode, si on peut accéder au code de la méthode, alors la dépendance de x est calculée : il est alors dépendant des variables de l'instruction *return* ainsi que des paramètres effectifs de la méthode meth.
Paramètres effectifs et formels	*Décl. :* void meth(x) *Appel :* meth(y)	x=depend(y) pcm(meth)=pc	Les paramètres formels dépendent des paramètres effectifs car leur valeur est une copie de celle des paramètres effectifs associés.
Table et compteur	T[y]=x	x=depend(y) T=depend(x)	Un élément d'une table dépend de son compteur, parce que le compteur peut révéler une information sur l'élément utilisé. Le tableau dépend de son contenu.
Initialisation d'objet	C x = new C(y)	x=depend(y) pcm(Constructeur)=pc	Un objet dépend des paramètres de ses constructeurs
Attributs d'objets	x.f1 = y x.f1 = y.f2	x=depend(y) f1=depend(y) f1=depend(f2)	Un objet dépend des variables affectées à ses attributs. Les attributs dépendent des variables qui leur sont affectées.
Méthodes d'objets	x = y.meth(z)	x=depend(y) x=depend(return(y)) return(meth)=depend(z) pcm(meth)=pc	Une variable dépend de l'instruction de retour de la méthode qui lui est affectée. Elle dépend également de l'objet à partir de quelle cette méthode est appelée. Si le code de la méthode n'est pas disponible, ses variables de retour dépendent de ses paramètres effectifs.

Flux implicites			
Condition	Cond ?Cons :Alt if Cond then Cons else Alt	pc = pc ⋃ Vars(Cond) ModifVars(Cons) = depend(pc) ModifVars(Alt) = depend(pc)	Toute modification des variables du conséquent ou de l'alternative peuvent donner une idée sur la valeur de la condition. C'est pourquoi les variables modifiée à l'intérieur de l'une de ces deux structures dépendent des variables dans la condition.

Itération	while (Cond)
	do (Loop)
	do (Loop)
	while (Cond)
	for (Cond)
	do (Loop)
		pc = pc U Vars(Cond)
		ModifVars(Loop) = depend(pc)
			Les variables modifiées dans le bloc *Loop* dépendent de ceux dans la condition.

Instruction "Selon "	Switch (Cond){
	case (val) : (Case) ;
	default : (Def) ;
	}
		pc = pc U Vars(Cond)
		ModifVars(Case) = depend(pc)
		ModifVars(Def) = depend(pc)
			Les variables modifiées dans le bloc *Case* ou *Default* dépendent de ceux dans la condition.

Opérateurs logiques	(Exp1) && (Exp2) (Exp1) || (Exp2)	pc = pc U Vars(Exp1) ModifVars(Exp2) = depend(pc) pc = pc U Vars(Exp2)	Dans les opérateurs doubles, l'évaluation de l'Exp2 dépend de la valeur de Exp1. Dans ce cas, il peut représenter un canal caché qui divulgue des informations secrète sur Exp2.

Annexe B

Modèle de génération des composants cryptographiques

Dans ce code, qui représente un template utilisé pour la génération du code d'implémentation des composants de cryptographie, le compilateur va remplacer les termes clefs placés entre "@" par leurs valeurs adéquates selon l'utilisation de ce composant dans le système.

Par exemple, le terme *@SecCompName@* sera remplacé par le nom de ce composant de sécurité, qui est composé du nom du composant fonctionnel associé, attaché au mot *security*. Le terme *@ServiceItfs@* contient l'ensemble des services que ce composant fournit, qui représentent les services initialement fournis par le composant fonctionnel qui nécessitent une opération cryptographique (chiffrement ou signature).

Notons que le code de cryptographie ne contient pas de termes variables, on peut donc facilement modifier l'algorithme de chiffrement ou de signature, ou même la méthode de chiffrement, pour utiliser le chiffrement symétrique au lieu de l'asymétrique, par exemple.

Le concepteur de sécurité pourra s'il le désire créer manuellement ses propres composants cryptographiques au lieu des composants générés par l'outil. Il devra les nommer et les placer de manière adéquate pour le déploiement.

Listing B.1 – Fichier SecurityImplTemplate

```
package @PackageName@;

import java.io.IOException;
import java.io.Serializable;
```

```
import javax.crypto.*;
import java.security.*;

import org.osoa.sca.annotations.Reference;

@Imports@

public class @SecCompName@ implements @ServiceItfs@{

@RefDecls@

    private static PublicKey pub;
    private static PrivateKey priv;

    static{
      generateKeys();
    }

    public @SecCompName@() {
        write("@SecCompName@ started...");
  }

    @ServiceItfImplems@

    /*************** Messages textes **********************/

  public String encryptString(String messString, SendKeyItf targetKeyRef){
    byte[] message = messString.getBytes();
      Cipher c=null;
      byte[] encrypted=null;

    //Lire la clef publique du serveur
      PublicKey pubKey = targetKeyRef.getKey();

    //Créer un cryptogramme pour le chiffrement asymétrique RSA
    //Initialiser le cryptogramme pour le chiffrement avec la clef publique
    //Chiffrer le message avec le cryptogramme
    try{
        c = Cipher.getInstance("RSA/ECB/NoPadding");
        c.init(Cipher.ENCRYPT_MODE, pubKey);
        encrypted = c.doFinal(message);

      } catch (java.security.NoSuchAlgorithmException e) {
```

```
        write_err("Create Asym Cipher - No such algorithm");
    } catch (javax.crypto.NoSuchPaddingException e) {
        write_err("Create Asym Cipher - No such padding");
    } catch (java.security.InvalidKeyException e) {
    write_err("Create Asym Cipher - Invalid key");
    } catch(javax.crypto.IllegalBlockSizeException e){
        write_err("Create Asym Cipher - Illegal block size : "+ c.
            getBlockSize());
    } catch (javax.crypto.BadPaddingException e) {
        write_err("Create Asym Cipher - Bad Padding");
    }
    return new String(encrypted);
}

public String decryptString(String encryptedString){

byte[] encrypted = encryptedString.getBytes();
    Cipher c=null;
    byte[] decrypted=null;

    //Créer un cryptogramme
//Initialiser le cryptogramme pour le déchiffrement avec la clef privée
//Déchiffrer le message avec le cryptogramme
    try{
      c = Cipher.getInstance("RSA/ECB/NoPadding");
      c.init(Cipher.DECRYPT_MODE, priv);
      decrypted = c.doFinal(encrypted);

    } catch(javax.crypto.NoSuchPaddingException e){
      write_err("Asym Cipher - No such padding");
    } catch(javax.crypto.IllegalBlockSizeException e){
      write_err("Asym Cipher - Illegal block size "+c.getBlockSize());
    } catch(javax.crypto.BadPaddingException e){
      write_err("Asym Cipher - Bad padding");
    } catch (java.security.NoSuchAlgorithmException e) {
      write_err("Asym Cipher - No such algorithm");
    } catch (java.security.InvalidKeyException e){
      write_err("Asym Cipher - Invalid key");
    }

    return new String(decrypted);

}
```

```
  public String signString(String messageString){

  byte[] message = messageString.getBytes();
  byte[] signed=null;
  Signature sig=null;



  //Obtenir et initialiser une instance de l'objet Signature
  //avec la clef privée
  try{
      sig = Signature.getInstance("MD5WithRSA");
      sig.initSign(priv);
    } catch (java.security.NoSuchAlgorithmException e){
      write_err ("Signature init - Algorithm not found");
    } catch (java.security.InvalidKeyException e){
      write_err ("Signature init - Invalid Key");
    }

    // Signer la donnée

  try{
      sig.update(message);
    signed = sig.sign();
    } catch (java.security.SignatureException e){
      write_err ("Signature of data - Signature?");
    }

  return new String(signed);
}


public boolean verifyString(String clearString, String signedString,
    SendKeyItf targetKeyRef){

    byte[] clear = clearString.getBytes();
    byte[] signed = signedString.getBytes();
    Signature sig=null;

    PublicKey pubKey = targetKeyRef.getKey();

    // Initialiser la signature
```

```
  try{
    sig = Signature.getInstance("MD5WithRSA");
    sig.initVerify(pubKey);
  } catch (java.security.NoSuchAlgorithmException e){
    write_err("Signature init - No Such algorithm");
  } catch (java.security.InvalidKeyException e){
      write_err("Signature init - Invalid key");
  }

  // Utiliser la donnée signée
  try{
    sig.update(clear);
  } catch(java.security.SignatureException e){
    write_err("Signature update - Signature?");
  }

  // Vérifier
  boolean verified = false;
  try {
      verified = sig.verify(signed);
  } catch (SignatureException se) {
    verified = false;
  }

  return verified;
}

/*************** Messages objets  **********************/

public Object encryptObject(Serializable message, SendKeyItf targetKeyRef
    ) {
    Cipher c = null;
    SealedObject encrypted = null;
    // Lire la clef publique du serveur
    PublicKey pubKey = targetKeyRef.getKey();

    //Créer un cryptogramme pour le chiffrement asymétrique RSA
    //Initialiser le cryptogramme pour le chiffrement avec la clef
       publique
    //Chiffrer le message avec le cryptogramme
    try {
        c = Cipher.getInstance("RSA");
```

```
            c.init(Cipher.ENCRYPT_MODE, pubKey);
            encrypted = new SealedObject(message, c);

    } catch (java.security.NoSuchAlgorithmException e) {
                write_err("Create Asym Cipher - No such algorithm");
    } catch (javax.crypto.NoSuchPaddingException e) {
                write_err("Create Asym Cipher - No such padding");
    } catch (java.security.InvalidKeyException e) {
                write_err("Create Asym Cipher - Invalid key");
    } catch (javax.crypto.IllegalBlockSizeException e) {
                write_err("Create Asym Cipher - Illegal block size: ");
    } catch (IOException e) {
                write_err("IO exception");
    }
    // Renvoyer le message chiffré au serveur

    return encrypted;
  }

  public Object decryptObject(Object encrypted) {

        Cipher c = null;
        Object decrypted = null;

    //Créer un cryptogramme
    //Initialiser le cryptogramme pour le déchiffrement avec la clef privée
    //Déchiffrer le message avec le cryptogramme
        try {
            c = Cipher.getInstance("RSA");
            c.init(Cipher.DECRYPT_MODE, priv);

            decrypted = ((SealedObject)encrypted).getObject(c);

    } catch (javax.crypto.NoSuchPaddingException e) {
            System.err.println("Asym Cipher - No such padding");
    } catch (javax.crypto.IllegalBlockSizeException e) {
            System.err.println("Asym Cipher - Illegal block size " + c.
               getBlockSize());
    } catch (IOException e) {
            System.err.println("Asym Cipher - IO Exception");
    } catch (java.security.NoSuchAlgorithmException e) {
            System.err.println("Asym Cipher - No such algorithm");
    } catch (java.security.InvalidKeyException e) {
```

```
            System.err.println("Asym Cipher - Invalid key");
    } catch (ClassNotFoundException e){
            System.err.println("Class not found");
    } catch (BadPaddingException e){
            System.err.println("Bad padding");
    }

        return decrypted;

}

    public SignedObject signObject(Serializable objectToSign) {
        SignedObject so = null;
      Signature sig = null;

      //Génération de la paire de clefs
      try {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

      // Génération d'un nombre aléatoire basé sur  SHA de SUN

      SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
      keyGen.initialize(1024, random);

    } catch (NoSuchAlgorithmException nsae) {
      write_err("Key Gen - No such algorithm");
    } catch (java.security.NoSuchProviderException e) {
      write_err("Key Gen - Provider not found");
    }

    // Obtenir et initialiser une instance de l'objet Signature
    //avec la clef privée

    try{
        sig = Signature.getInstance("MD5WithRSA");
      so = new SignedObject(objectToSign, priv,sig);

      } catch (java.security.NoSuchAlgorithmException e){
        write_err ("Signature init - Algorithm not found");
      } catch (java.security.InvalidKeyException e){
        write_err ("Signature init - Invalid Key");
      } catch (java.security.SignatureException e){
        write_err ("Signature of data - Signature?");
```

```
    }catch (java.io.IOException e){
      write_err ("Signature - io problem");
    }
    return so;
}

public Object verifyObject(Object signed, SendKeyItf targetKeyRef) {
  Object verified = null;
  PublicKey pub = targetKeyRef.getKey();
    Signature sig = null;

    try{
      sig = Signature.getInstance("MD5WithRSA");
      if (((SignedObject)signed).verify(pub, sig)){
        verified = ((SignedObject)signed).getObject();
      }else{
        System.err.println("Invalid Signature!!!");
        System.exit(-1);
      }

    } catch (java.security.NoSuchAlgorithmException e){
      write_err("Verification - No Such algorithm");
    } catch (java.security.InvalidKeyException e){
      write_err("Verification - Invalid key");
    } catch (java.lang.ClassNotFoundException e) {
      write_err("Verification - Class not found");
    }catch (java.io.IOException e){
      write_err ("Verification - io problem");
    }catch (java.security.SignatureException e){
      write_err ("Verfication - signature problem");
    }

        return verified;
}

public static void generateKeys() {

      try {
          KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

          SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN")
              ;
          keyGen.initialize(1024, random);
```

```
            // Créer une paire de clefs
            KeyPair pair = keyGen.generateKeyPair();
            priv = pair.getPrivate();
            pub = pair.getPublic();

    } catch (NoSuchAlgorithmException nsae) {
            System.err.println("--- Client Security : Key Gen - No such
                algorithm");
    } catch (java.security.NoSuchProviderException e) {
      System.err.println("--- Client Security : Key Gen - Provider not found"
          );
    }

  }

  public PublicKey getKey() {
        return pub;
  }

    /**  * * Méthodes utilitaires  */
    private void write(String message) {
        System.out.println("--- @SecCompName@ : " + message + " ---");
  }

    private void write_err(String message) {
        System.err.println("--- @SecCompName@ : " + message + " ---");
  }

}
```

Listing B.2 – Fichier SendKeyItfTemplate

```
package @PackageName@;

import java.security.*;

public interface SendKeyItf {
  public PublicKey getKey();
}
```

Listing B.3 – Fichier SendItfTemplate

```
package @PackageName@;

public interface SendItf_@RefName@_@ServiceName@ {

@Implem@

}
```

Annexe C

Représentation des étiquettes dans Maude

Ceci est une proposition pour représenter les étiquettes dans Maude en utilisant le modèle d'étiquettes à base de jetons présenté dans la section III.2.2, partie I.

Listing C.1 – Représentation en Maude des étiquettes à base de jetons

```
mod LABEL is
   sorts Tag TagList Label .
   subsort Tag < TagList .

  ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Tag .
  op nil : -> TagList .
  op _,_ : TagList TagList -> TagList [assoc prec 60 id: nil] .
  op `{_;_`} : TagList TagList -> Label .

  op confProj : Label -> TagList .
  op integProj : Label -> TagList .

  op leq : Label Label -> Bool .
  op _leqConf_ : TagList TagList -> Bool .
  op _leqInteg_ : TagList TagList -> Bool .
  op _occursIn_ : Tag TagList -> Bool .

  vars Tl Tl' : TagList . vars T T' : Tag . vars L L' : Label .


***Equations
  eq confProj({ Tl ; Tl' }) = Tl .
```

```
  eq integProj({ Tl ; Tl' }) = Tl' .

  eq leq(L,L') = (confProj(L) leqConf confProj(L')) and (integProj(L)
     leqInteg integProj(L')) .

  eq (T) leqConf nil = false .
  eq nil leqConf T = true .
  eq nil leqConf nil = true .
  ceq (T) leqConf (Tl') = true if T occursIn Tl' .
  ceq (T,Tl) leqConf (Tl') = false if (T occursIn Tl' == false) .
  crl[nextConf] : (T,Tl) leqConf (Tl') => (Tl) leqConf (Tl') if (T occursIn
     Tl') .

  eq nil leqInteg (T) = false .
  eq T leqInteg nil = true .
  eq nil leqInteg nil = true .
  ceq Tl leqInteg (T') = true if T' occursIn Tl .
  ceq (Tl) leqInteg (T',Tl') = false if (T' occursIn Tl == false) .
  crl[nextInteg] : (Tl) leqInteg (T',Tl') => (Tl) leqConf (Tl') if (T'
     occursIn Tl) .

  eq T occursIn nil = false .
  eq T occursIn (T',Tl) = if T == T' then true else T occursIn Tl fi .

endm
```

Annexe D

Primitives Cryptographiques de Java

Le langage de programmation Java comprend plusieurs aspects facilitant la programmation sécurisée.

- Pas de pointeurs, ce qui veut dire qu'un programme Java ne peut pas adresser les emplacements en mémoire arbitrairement dans l'espace d'adressage.
- Un vérificateur de *bytecode*, qui opère une fois la compilation terminée sur les fichiers *.class* et vérifie les problèmes de sécurité avant l'exécution. Par exemple, une tentative d'accéder à un élément de tableau se trouvant au delà de la taille d'un tableau sera rejetée.
- Un contrôle très précis sur l'accès aux ressources pour les applets et les applications. Par exemple, un applet peut être empêché d'accéder à un espace disque ou peut être autorisé de lire à partir d'un répertoire en particulier. Ces configurations sont définies dans un fichier *java.policy*.
- Un grand nombre de bibliothèques sont disponibles permettant d'appliquer les mécanismes cryptographiques les plus connus. Nous présentons dans ce qui suit des exemples de primitives que nous utilisons dans les composants cryptographiques insérés dans le système.

Les bibliothèques utilisées sont JCE (*Java Cryptography Extension*) et JSSE (*Java Secure Sockets Extension*).

Empreinte numérique La classe *MessageDigest* manipule les empreintes numériques. Le code D.1 montre un exemple de son utilisation.

Listing D.1 – Exemple d'utilisation de l'empreinte numérique en Java

```
String plainText = "Texte en clair";
// Créer une empreinte numérique en utilisant l'algorithme MD5
MessageDigest messageDigest = MessageDigest.getInstance("MD5");
// Calculer et afficher l'empreinte
messageDigest.update(plainText.getBytes("UTF8"));
System.out.println(new String(messageDigest.digest(), "UTF8"));
```

Chiffrement à clef privée La classe *Cipher* manipule les algorithmes à clefs privées en utilisant une clef produite par la classe *KeyGenerator*. La portion de code D.2 montre un exemple de chiffrement en utilisant l'algorithme de chiffrement à clef privée DES.

Listing D.2 – Exemple d'utilisation du chiffrement à clef privée en Java

```
String plainText = "Texte à encoder" ;
//Génération de la clef DES
KeyGenerator keyGen = KeyGenerator.getInstance("DES"); keyGen.init(56);
Key key = keyGen.generateKey();
//Début du chiffrement
Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] cipherText = cipher.doFinal(plainText.getBytes("UTF8"));
//Début du déchiffrement
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] newPlainText = cipher.doFinal(cipherText);
System.out.println( new String(newPlainText, "UTF8") );
```

Chiffrement à clef publique De même que pour le chiffrement à clef privée, le chiffrement à clef publique est fait en utilisant la classe *Cipher*. Mais comme le chiffrement à clef publique utilise une paire de clefs, la classe *KeyPairGenerator* est utilisée pour les générer. La portion de code D.3 montre un exemple de chiffrement avec l'algorithme RSA.

Listing D.3 – Exemple d'utilisation du chiffrement à clef publique en Java

```
String plainText = "Texte à encoder" ;
//Génération de la paire de clefs RSA
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(1024);
KeyPair key = keyGen.generateKeyPair();
//Début du chiffrement
```

```
Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());
byte[] cipherText = cipher.doFinal(plainText.getBytes("UTF8"));
//Début du déchiffrement
cipher.init(Cipher.DECRYPT_MODE, key.getPrivate());
byte[] newPlainText = cipher.doFinal(cipherText);
System.out.println( new String(newPlainText, "UTF8") );
```

Signature digitale Java utilise la classe *Signature* pour signer et vérifier les messages. La portion de code D.4 montre un exemple de signature avec l'algorithme MD5/RSA. Comme RSA est utilisé, la génération de la paire de clefs est faite avec la classe *KeyPairGenerator* comme illustré dans le paragraphe précédent.

Listing D.4 – Exemple d'utilisation de la signature digitale en Java

```
//Déclaration des variables
PublicKey pub;
PrivateKey priv;
byte[] message = messageString.getBytes();
byte[] signed=null;
Signature sig,sig2=null;
//Initialisation des clefs publique et privée avec KeyPairGenerator
...
//Initialisation de l'objet Signature pour signer le message
sig = Signature.getInstance("MD5WithRSA");
sig.initSign(priv);
// Signature du message
sig.update(message);
signed = sig.sign();
//Initialisation de l'objet Signature pour vérifier le message
sig2 = Signature.getInstance("MD5WithRSA");
sig2.initVerify(pubKey);
//Vérification
sig2.update(message);
boolean verified = sig.verify(signed);
```

Certificats Java fournit des certificats et des clefs publiques provenant de plusieurs CAs. Il supporte le standard *X.509 Digital Certificate*.

La plateforme Java utilise un *keystore* comme dépôt pour les clefs et les certificats. Physiquement, le *keystore* est un fichier (qui peut être chiffré) avec le nom par défaut *.keystore*. Les

clefs et les certificats peuvent avoir des noms, appelés *alias* et chaque alias peu être protégé par un mot de passe unique. Le *keystore* lui-même est protégé par un mot de passe.

La plateforme Java utilise un *keytool* pour manipuler le *keystore*. Cet outil peut être utilisé pour exporter une clef dans un fichier en format X.509, qui peut être signé par une autorité de certification et réimporté dans le *keystore*.

Il existe également un *keystore* particulier utilisé pour stocker les certificats de l'autorité de certification, qui contiennent à leur tour les clefs publiques pour la vérification de la validité des autres certificats. Ce *keystore* est appelé *truststore*. Java contient un *truststore* par défaut dans un fichier appelé *cacerts*.

L'objectif de ce travail est de fournir des modèles et outils pour simplifier la construction des systèmes distribués à base de composants sécurisés, ainsi que la gestion des propriétés de sécurité, en utilisant des outils de haut niveau d'abstraction pour la configuration et la reconfiguration dynamique.
En plus des propriétés d'accessibilité et de communications sécurisées classiques, nous focalisons notre travail sur une propriété des systèmes répartis plus générale : la non-interférence. Cette propriété atteste qu'il ne doit pas y avoir de flux d'information entre des parties publiques et privées du système. Ce qui implique le suivi de l'acheminement de l'information entre les différentes composantes du système distribué.
Notre objectif principal est donc de proposer un modèle, accompagné d'un ensemble d'outils, garantissant la propriété de la non-interférence *à la construction* du système, et ce à une plus grosse granularité : celle des composants. Ces outils permettent de (1) configurer les paramètres de sécurité des composants et des liaisons entre eux, (2) vérifier la propriété de non-interférence dans le code d'un composant et entre les différents composants du système et (3) générer automatiquement le code nécessaire pour appliquer ces propriétés de sécurité. D'autre part, nous proposons une architecture permettant de vérifier *dynamiquement* la propriété de non-interférence dans un système réparti.

Mots clés : Systèmes répartis/distribués, non-interférence, systèmes à base de composants, génération de code.

The goal of this thesis is to provide models and tools to simplify secured component-based distributed systems' construction and the management of their security properties, by using high-level tools for dynamic configuration and reconfiguration.
In addition to the classic properties of accessibility and secured communications, we focus on a more general security property of distributed systems : the non-interference. This property says that there mustn't be information flow between secret and public parts of the system ; which requires information flow control across the system.
Our main objective is to propose a model and set of tools guarantying the non-interference property *at compile-time*, and at a bigger granularity : the components. These tools are (1) tools for configuring security parameters of components and binding between components, (2) a compiler checking the non-interference property, and (3) tools for automatic generation of code assuring these security properties. On the other hand, we present an architecture enabling a *dynamic* verification of the non-interference property in a distributed system.

Keywords : Distributed systems, non-interference, component-based software engineering, code generation.